Automatically Granted Permissions in Android apps

An Empirical Study on their Prevalence and on the Potential Threats for Privacy

Paolo Calciati
IMDEA Software Institute
Spain

Konstantin Kuznetsov
CISPA Helmholtz Center for Information Security
Germany

Alessandra Gorla
IMDEA Software Institute
Spain

Andreas Zeller
CISPA Helmholtz Center for Information Security
Germany

ABSTRACT

Developers continuously update their Android apps to keep up with competitors in the market. Such constant updates do not bother end users, since by default the Android platform automatically pushes the most recent compatible release on the device, unless there are major changes in the list of requested permissions that users have to explicitly grant. The lack of explicit user's approval for each application update, however, may lead to significant risks for the end user, as the new release may include new subtle behaviors which may be privacy-invasive. The introduction of permission groups in the Android permission model makes this problem even worse: if a user gives a single permission within a group, the application can *silently request further permissions in this group with each update*—without having to ask the user.

In this paper, we explain the threat that permission groups may pose for the privacy of Android users. We run an empirical study on 2,865,553 app releases, and we show that in a representative app store more than ∼17% of apps request at least once in their lifetime new *dangerous* permissions that the operating system grants without any user's approval. Our analyses show that apps *actually use* over 56% of such automatically granted permissions, although most of their descriptions do not explicitly explain for what purposes. Finally, our manual inspection reveals clear abuses of apps that *leak sensitive data* such as user's accurate location, list of contacts, history of phone calls, and emails which are protected by permissions that the user never explicitly acknowledges.

ACM Reference Format:
Paolo Calciati, Konstantin Kuznetsov, Alessandra Gorla, and Andreas Zeller. 2020. Automatically Granted Permissions in Android apps: An Empirical Study on their Prevalence and on the Potential Threats for Privacy. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3379597.3387469

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387469

1 INTRODUCTION

In the early releases of the Android OS, users had to accept *all* permissions requested by an app upfront to install and use it. Moreover, users had to approve *every new app release* before installing it on their device if the update included *any* new dangerous permission request. For years the research community heavily criticized this permission model and pointed out its main flaws: app developers could easily request more permissions, and users were willing to grant them in order to use the app. Moreover, most users did not understand the implications of granting dangerous permissions to apps [23]. Thus, asking users to explicitly approve each release which involved a change in the list of requested permissions had two effects: either users approved the change without worrying about the implications, or they did not approve it keeping an old version of the app on the device, augmenting the challenges for app developers who had to assume users could use different releases of their product.

Given daily or weekly frequency of updates and potential changes in the list of permission requests [17], most app developers had the habit of asking for extra permissions in first releases. The rationale was that even if early releases were not using some of the requested permissions, the user had to grant them anyway to use the app, and developers could use these permissions in later releases without bothering the user for further approval. Several studies show that this practice caused many apps to be *overprivileged* and left space for potential security attacks [20].

With the release of Android 6, Google significantly revised the permission model of Android apps. Apps now *dynamically* ask for permissions, i.e., they ask for user's consent when they use the permission for the first time. This change has the positive effect that users can now understand when and how an app needs to access a protected resource and can decide to deny such permission. Secondly, to simplify the permission model for user understanding, Android now uses *permission groups*. With this change, users would be asked to grant a permission only if the app asks for a permission that does not belong to any permission group that has been already approved before. This means that, for instance, if an app already had the granted permission to access the coarse location of the user, it may access in later releases the fine-grained location information (obtained through GPS) without further consent.

While introducing permission groups clearly improves usability, making them preferred by the majority of users [3] over the previous model, it introduces more threats for their privacy and security, since behavior changes are more subtle to notice. Some empirical

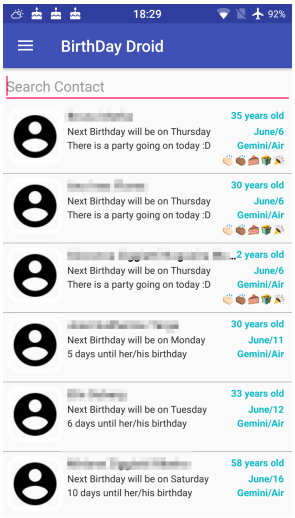

Figure 1: User interface of the BirthDayDroid app

studies show that apps tend to leak more information over time, even when there is no change in the list of permissions [14, 34]. Other studies show that apps can leak sensitive information protected by dangerous permissions even without asking for that permission [33]. We argue that, on top of the risks already highlighted by other studies, permission groups pose a significant risk for the security and privacy of users, since developers can push a new release to the user's device and may access resources that were not explicitly granted.

To properly explain the threat scenario, we use a real open source application as motivating example. The BirthDayDroid app from F-droid[1] looks as shown in Figure 1. This app helps users remember birthdays of contacts. To this end, it scans the contact list on the device to collect birthday dates. It uses this information to show the age of each contact and the days left until the next birthday. The app obviously requires the READ_CONTACTS permission in order to work, which belongs to the contacts permission group. To demonstrate the threat scenario, we update the app, simulating a malicious developer, and include a payload that accesses the list of all the accounts (i.e., Facebook, Twitter, Google, etc), writes them to a log file, and finally sends them to one of our servers. To do so, the new release of the app requires the GET_ACCOUNTS permission. Although this permission appears in the new manifest, as shown in Figure 2, the end user would likely not notice it, as the Android framework grants this permission automatically given that it belongs to the same permission group of the already present READ_CONTACTS. *With an automatic update on the user's device, the app can covertly collect sensitive data on the user's accounts.*

In this paper, we study how real Android apps evolve in their permission requests, and we show the implications of permission groups on the user's privacy. We run an empirical study on a large dataset of over 2 million Android apps. We show that many apps initially ask for some permissions and later ask for more privacy-invasive permissions that the underlying OS automatically grants.

[1]https://f-droid.org/en/packages/com.tmendes.birthdaydroid/

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.tmendes.birthdaydroid">

  <uses-permission android:name="android.permission.GET_ACCOUNTS"/>
  <uses-permission android:name="android.permission.READ_CONTACTS"/>
  <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
  <uses-permission android:name="android.permission.VIBRATE"/>
  <uses-permission android:name="android.permission.WAKE_LOCK"/>
  <uses-permission android:name=
                      "android.permission.RECEIVE_BOOT_COMPLETED"/>
  ...
</manifest>
```

Figure 2: Excerpt of the `AndroidManifest.xml` file of the modified BirthDayDroid app

We show that most apps *actually use* such automatically granted permissions, and in many of the cases that we manually inspect they leak new sensitive information without end-user could easily notice. The paper has the following contributions:

- It reports for the first time a real threat for the privacy and security of Android users due to permissions groups.
- It presents an empirical study on 2,865,553 Android binary files and shows that in a representative app store over ~17% of the apps request dangerous permissions that the OS automatically grants without explicit user's approval.
- We manually inspect some of the most suspicious cases, and we report clear abuses of apps that leak sensitive data such as user's accurate location, list of contacts, history of phone calls, and emails.

The remainder of the paper is structured as follows: Section 2 explains the Android permission model and how it changed from Android 6 up the current release. We thus clearly present the threats of automatically granted permissions, from Android 6 up to the upcoming release of the Android OS. Section 3 presents our empirical study on the prevalence of automatically granted permissions in two large datasets. Section 4 shows the results of our preliminary static analysis to understand if apps actually use automatically granted permissions. Section 5 discusses in details the most interesting cases of clear information leaks. Section 6 presents the related work. Section 7 reports the limitations and threats to validity for our study, and Section 8 concludes the paper summarizing the main findings.

2 PERMISSION MODEL AND PERMISSION GROUPS IN ANDROID

Android protects the access to sensitive user data and critical system features by means of permissions: when an app wants to access such data or wants to use such features, it must request the corresponding permission first. Depending on the permission, the system might grant it automatically, such as when an app wants to connect to the Internet, or might ask the user to explicitly approve it. *Dangerous* permissions require explicit user agreement, and the Android system asks the user to grant them in different ways, depending on the version of the system installed on the mobile device [2].

[2]https://developer.android.com/guide/topics/permissions/overview

Old Android permission model. If the device runs Android 5.1.1 or lower—corresponding to API level 22 or lower—the system asks the user to grant all dangerous permissions to the application at installation-time. If the user accepts, the system grants all requested permissions. By declining, instead, the user cannot use nor install the application. Moreover, the user has to explicitly accept every new permission requested with following updates of the app. Without approval, the update process does not even start. Despite working only on quite obsolete versions of the Android system (i.e., pre Android 6.0), this permission model still runs on ~25% of the Android devices [3].

Applications targeting devices running Android 6.0 (API level 23) or higher, which accounts for ~75% of Android devices, must ask for user's approval before using a permission for the first time (or if the user has revoked it before), otherwise they would crash. The Android system shows a system dialog specifying which permission group the app wants to access and asks the user to grant or deny such permission. Developers can optionally add a custom explanation message to help users understand why the app needs that specific permission.

Permission groups since Android 6.0. To help users make informed choices about granting or not a permission without overwhelming them with too many requests, the Android system organizes dangerous permissions related to similar functionalities into groups. Granting permission to a group implies that an app can include in the manifest file any other permission of that group in future releases, and the Android OS would grant it automatically.

Table 1 shows the dangerous permissions and their relative groups in use in Android 6.0, which is the first release of Android using this permission model. As the Table shows, these groups have very broad functionalities.

For instance, both RECEIVE_SMS and SEND_SMS permissions belong to the SMS group, as both permissions operate on SMSs. When an app requests a new dangerous permission, the system prompts the request dialog only if the app does not have any permission in the same permission group already granted. If an application already has the RECEIVE_SMS permission granted, and in a new release it requests the SEND_SMS permission, the latter permission would be automatically granted by the system without the user even noticing it.

This could allow malicious developers to exploit the permissions already granted to an app to automatically gain access to sensitive user data without them even noticing it, as we show in Section 1.

Android 7 (API level 24 and 25) does not add any new dangerous permissions or permission groups. As of August 2019, Android 6.0 and 7.0 are the most prevalent releases among all Android devices, accounting to over 36% of the market share.

Recent changes to permission groups. With the release of Android 8 (API level 26), Google partially addresses the concerns we raise in this paper by introducing two new dangerous permissions to the PHONE group: READ_PHONE_NUMBERS and ANSWER_PHONE_CALLS. These two permissions allow an application to answer incoming phone calls programmatically, via the acceptRingingCall() API, and to access the phone numbers stored in the device, respectively. Additionally, Android 8 introduces a number of changes on how some of the current permissions work. A Wi-Fi scan requires that the app has either the CHANGE_WIFI_STATE permission, or any LOCATION permission. Moreover, applications can no longer access user accounts unless either the authenticator owns the account or the user explicitly grants the access.[4] However, our experiments show that on a device running Android 8.1 it is still possible to access a list of available accounts without user's consent, if an app targets older version of the API.

Android 9 (API level 28) introduces further updates to protect the user's personal data. First and foremost, there is now a separate CALL_LOG permission group, and all the permissions related to calls belong to this group. Separating these permissions from others related to the PHONE group partially reduces the risk that an app with the READ_PHONE_STATE permission—used, for instance, to read the unique ID of the device—could *also* make phone calls without explicitly asking for this permission.

This Android release also introduces a new dangerous permission in the PHONE group: ACCEPT_HANDOVER. It allows a calling app to continue a call which was started in another app.

Apps running on Android 9 can no longer read phone numbers from the phone state data without having the READ_CALL_LOG permission. Moreover, in order to access the SSID and BSSID values

Table 1: Dangerous permissions and their related groups in Android 6.0

Permission Group	Permissions
CALENDAR	READ_CALENDAR WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS WRITE_CONTACTS GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE CALL_PHONE READ_CALL_LOG WRITE_CALL_LOG ADD_VOICEMAIL USE_SIP PROCESS_OUTGOING_CALLS
SENSORS	BODY_SENSORS
SMS	SEND_SMS RECEIVE_SMS READ_SMS RECEIVE_WAP_PUSH RECEIVE_MMS
STORAGE	READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE

[3] https://developer.android.com/about/dashboards visited in August 2019

[4] https://developer.android.com/about/versions/oreo/android-8.0-changes

Table 2: New or changed permission groups in Android 8 (marked with (*)) and 9.

Permission Group	Permissions
CALL_LOG	READ_CALL_LOG, WRITE_CALL_LOG, PROCESS_OUTGOING_CALLS
PHONE	READ_PHONE_STATE READ_PHONE_NUMBERS(*), CALL_PHONE, ANSWER_PHONE_CALLS(*), ADD_VOICEMAIL, USE_SIP, ACCEPT_HANDOVER

returned from the Wi-Fi scan getConnectionInfo() method, an app must now explicitly request i) any LOCATION permission, ii) ACCESS_WIFI_STATE permission, and iii) location services enabled on the device.

In October 2018 the Google Play store announced[5] a new policy for applications which deal with sensitive data covered by the SMS and CALL_LOG permission groups[6]. Apps can only use such permission groups if they are actively registered as the default SMS, Phone, or Assistant handler, and shall stop using them when they are no longer the default handler. However, this restriction does not affect other third-party stores or the way Android OS runs apps. Table 2 lists the new permission groups and summarizes the changes to existing ones introduced in Android 8 and 9 compared to Table 1.

Threats of Permission Groups despite Recent Android Releases. The recent changes in the permission model affecting Android 8 and 9 are an indicator that the previous model was too permissive. Allowing apps with READ_PHONE_STATE to make phone calls, or to read or write the call log, as still allowed in Android 6 and 7, is extremely risky and very counter-intuitive for users.

Despite the changes introduced in recent Android versions, the threats of granting permissions to groups rather than single permissions is still present, as we will show in the remainder of the paper. The assumption that permissions belonging to the same group have similar functionalities or at least operate on the same critical data, and therefore it is enough to ask user's consent only once for all permissions in the same group, opens up the possibility for malicious developers to gather more sensitive information without the user noticing it.

3 PREVALENCE OF AUTOMATICALLY GRANTED PERMISSIONS

We aim to assess how Android applications use (or abuse) permission groups in the new permission model. Specifically, we aim to assess if and how often apps request permissions that the Android system would automatically grant, and we aim to analyze how they use such permissions. For this study, we need multiple releases of the same app, and we need to focus only on releases that target the Android platform 6.0 or above, as explained in Section 2.

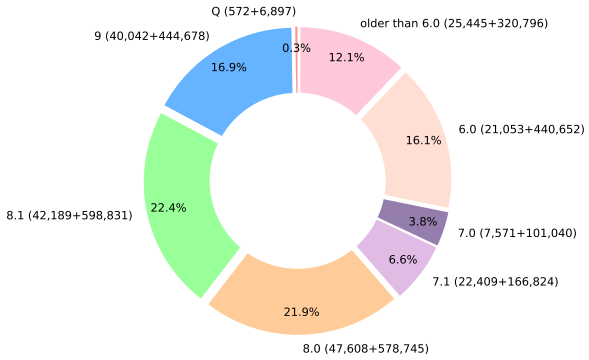

Figure 3: Android release distribution of our dataset (ApkPure and AndroZoo)

For each app binary, we check the list of requested permissions, and we compare it with the closest previous release of the same app in our dataset. For each newly requested permission, we verify whether any permission in the same group is present in the previous release, and we mark it as automatically granted by the underlying system. This, essentially, simulates the scenario described with the BirthDayDroid app in Section 1, where users first install a release of the app, grant all the permissions it requires, and later receive an update of the app with a new list of permissions that the operating system grants without asking for any further user's approval. We then further analyze all releases that include at least one automatically granted permission, and we assess if the app indeed accesses the information protected by the such permission, and if so whether it leaks this sensitive information.

We describe how we collect the dataset for our study in Section 3.1, and we explain how we design the study to quantify the risks of automatically granted permissions in Section 3.2. We finally present the results in Section 3.3.

3.1 Dataset Selection

To quantify the prevalence of automatically granted permissions in the Android ecosystem, we aim to find and analyze a substantial number of Android applications. Our analysis requires at least two releases of the same application to compare the list of permissions.

Since crawling the Google Play store for several releases would take quite a long time, we excluded crawling the official store. Although the authors of [7] report that the Google Play API can be queried for lower version codes, we could not reproduce this technique.

To obtain a representative dataset of the whole Android ecosystem we resort to two sources: ApkPure and Androzoo.

ApkPure is a third-party store that constantly adds new releases and removes outdated ones (it keeps at most 30 releases per app, available to users). We consider all categories defined in the store, excluding only games. (We exclude games since binaries of this

[5]https://android-developers.googleblog.com/2018/10/providing-safe-and-secure-experience.html
[6]https://play.google.com/about/privacy-security-deception/permissions/

category are usually very large, as they include a lot of graphics and data files. Their size can be up to 2GB, and they significantly impact the scalability of a large empirical study.) For each other category we crawl apps with publication date later than the official Android 6 release (05 October 2015). This is how we retrieve 216,492 APKs related to 13,109 different apps for later analysis.

The largest fraction of our dataset comes from Androzoo [2], a repository of Android apps maintained by the University of Luxembourg containing over 9,6 million APKs. AndroZoo is the result of many years of efforts of crawling the Google Play and similar Android stores to download as many apps as possible for research studies. For our purpose we consider only apps crawled from the Google Play Store, with a release date subsequent to the Android 6 release, and with at least two releases available. Following this process we retrieve 3,274,192 APKs, related to 794,795 apps.

ApkPure is a representative Android store of *recent* apps that users can safely install on their device. Androzoo, instead, is not a real app store that users can use, but on the other hand is representative of the Android ecosystem and its evolution.

3.2 Experiment Protocol

Android applications are distributed via APK package files, which are essentially zip format-type archives. For our study, we need to extract the list of automatically granted permissions of each release, stored in AndroidManifest.xml file along with other metadata. Despite being a simple and efficient analysis, downloading each APK, unpacking it and parsing the manifest of over 3 million apps would have been extremely time consuming. Our preliminary study showed that the Manifest file is usually located at the beginning of the archive. Thus, instead of downloading the whole package of each APK, which could be even hundreds of megabytes large, we initially download only the first 100Kb of the package. Next we parse the partially downloaded binary file using local file headers, extract AndroidManifest.xml if available, and decode the manifest file into a normal XML file with help of Androguard[7]. We then parse the XML file and collect the target release of Android SDK, version name and version code, the list of app components and the list of requested permissions. We finally compare this list with the permissions requested by the closest previous release. If the APK requests any new permission that the Android OS would automatically grant, based on the target Android release and the permission group listed in Section 2, we download the whole binary for further analyses. We discard it otherwise.

Downloading the first 100KB of the APK file does not guarantee that we download the Android manifest. Indeed, out of the ∼3 million APKs that we analyze with this heuristic, we could not analyze ∼12% of the APKs. We discard them for simplicity, obtaining a final dataset of 2,865,553 APKs.

Figure 3 plots how our aggregated dataset is split into different target Android releases. For each fraction we report the Android release that apps target (according to the Android manifest data) and the absolute number of APKs for the ApkPure and AndroZoo datasets (first and second number in parenthesis). For our analysis we discard APKs targeting releases prior to Android 6, since they use the old permission model.

[7] https://github.com/androguard/androguard

3.3 Results

Our analysis reports that on Androzoo there are 55,442 applications with at least one automatically granted permission. Specifically, we flag 63,970 single APKs with at least one automatically granted permission (our dataset includes more than one APK per application). This amounts to ∼7% of apps of the whole dataset of Androzoo of the apps targeting Android 6 or above.

These numbers are even more worrisome for the ApkPure dataset. Our analysis reports that there are 2,135 applications, and 2,834 single APKs, with at least one automatically granted permission. This amounts to ∼17% of apps of the whole dataset of ApkPure of the apps targeting Android 6 or above.

On a representative Android store, over ∼17% of the apps request at least once in their lifetime a dangerous permission that the OS can automatically grant without any user's approval.

Figures 4 and 5 give an insight into the results of our analysis on the two datasets. Each plot shows the amount of automatically granted permissions on the right-hand side, and the enabler permissions (on the left-hand side) that appear in the previous release. For instance, the plot shows that for the ApkPure dataset there are 148 applications that initially request the ACCESS_COARSE_LOCATION permission and later ask, and obtain without further user's approval, the ACCESS_FINE_LOCATION permission, which gives more accurate localization. Some permissions that are automatically granted may be enabled by different permissions in the same group. This is the case, for instance, for WRITE_CONTACTS, which is automatically granted in 104 apps in ApkPure. Roughly half of these apps initially request access to READ_CONTACTS and later request and automatically obtain the permission to write the contacts list, while the remaining apps initially requested the GET_ACCOUNTS permission from the same group.

Looking into the results, we see that 1,322 apps in ApkPure and 39,737 apps in AndroZoo request permissions to READ_STORAGE when they already have the WRITE_STORAGE permission. Similarly, 293 apps in ApkPure and 3,748 apps in AndroZoo request permissions to ACCESS_COARSE_LOCATION when they already have the ACCESS_FINE_LOCATION permission. These requests do not make much sense, as apps with permissions to write on the storage can implicitly read as well. Similarly, apps with permissions to access the GPS implicitly have the permission to access the coarser location. We exclude these automatically granted permissions from the plot, as they are implicitly granted for other reasons.

Though slightly different in terms of fractions, both plots show a very worrisome reality of the currently used permission model in Android apps: apps obtain without mandatory user's approval many dangerous permissions, such as the ability to make phone calls, and even worse the ability to access sensitive data, such as user accounts, contacts information, content of SMSs, and precise user's location. This sensitive information can be used for different purposes, and above all it can be leaked to third-party servers for advertisements.

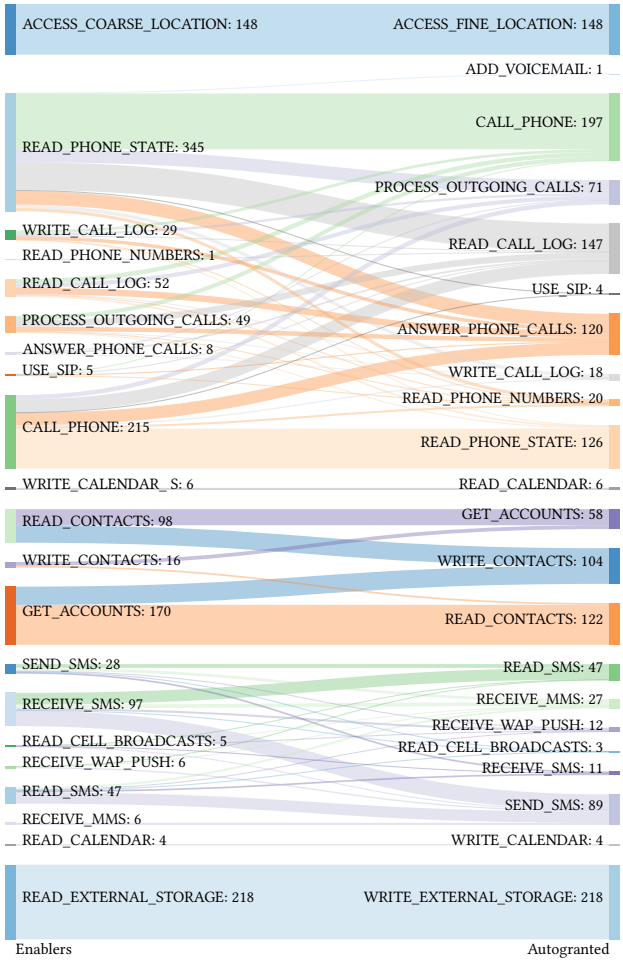

Figure 4: Automatically granted permissions in ApkPure

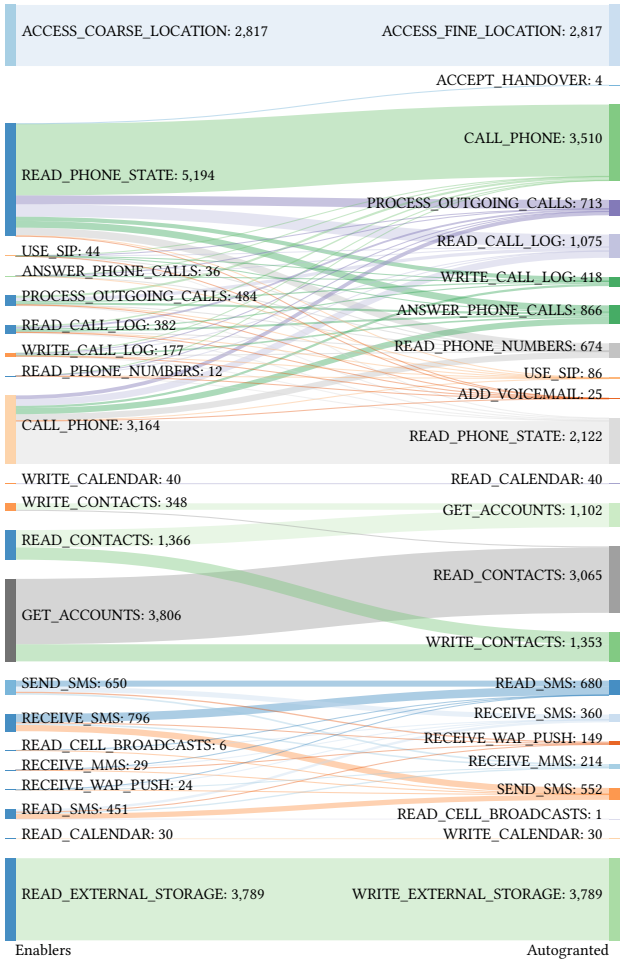

Figure 5: Automatically granted permissions in AndroZoo

Over 50% of automatically granted permissions allow apps to access sensitive data such as the list of contact, list of phone calls and the precise user's location. Over 40% of the automatically granted permissions allow apps to make phone calls or allow write operations on protected resources.

Given the prevalence of automatically granted permission, the next research question comes naturally. What do apps do with such permissions?

4 ACTUAL USE OF AUTOMATICALLY GRANTED PERMISSIONS

Given the prevalence of automatically granted permissions, as shown in Section 3, we want to assess if apps actually use these permissions and how. We thus statically analyze the bytecode of the application to detect uses of an automatically granted permission. We now explain how we implement this analysis, and we later present the results.

4.1 Static Bytecode Analysis

We resort to Soot [42] and FlowDroid [4] to statically analyze the whole package at the bytecode level to assess if the app actually uses each automatically granted permission. Android apps can access permission-protected resources in three different ways: 1) by calling specific Android API methods—e.g., access location with `getLastKnownLocation()`; 2) by querying databases with `ContentResolver`—e.g., retrieve the contact list using the `query("content://contacts")` method; and 3) by sending or receiving `Intents`—e.g., send the `android.intent.action.CALL` intent to make a phone call. Our analysis covers them all.

Specifically, we parse the bytecode and look for invocations of methods of the Android API that are permission-protected. There are a few permission-method mappings available from previous studies, such as PScout [5], Axplorer [8], DPSPEC [10]. Unfortunately, they all cover old Android versions (up to Android 7). We thus combine the lists of Axplorer and DPSPEC as a basis and extend it with new entries, which we found by analyzing the official

Android documentation [8] and the AOSP source code [9]. For each dangerous permission we thus have a list of Android API calls, intents, and content resolver URIs, protected with this permission. We then use Soot to parse each instruction in the Dalvik bytecode, looking for invocations related to the automatically granted permissions.

In order to understand whether these instructions can actually be executed (i.e., some of them may belong to functionalities of third-party components that apps do not use), we perform a reachability analysis of such statements. Starting from the method containing the statement of interest, we traverse the callgraph with a backward analysis and collect all methods in the call chain. The main Android components (activities, services, broadcast receivers, and data providers), which are declared in the Android manifest, are the only entry points of the app. If a call chain starts from any of the life-cycle or callback methods of these entry points, we assume that the permission protected statement is reachable and can be used by the app. Broadcast receivers, which are responsible for the interception of system broadcast messages, can be registered both in the code and in the Android manifest. If a receiver declares in the manifest an intent filter capturing a protected intent, we assume that it is further used, and thus the app accesses protected information bundled with it.

We thus analyze each APK with at least one automatically granted permission, and we label each permission as follows:

- If the app contains at least one statement protected by the corresponding automatically granted permission, and our analysis reports this code as reachable, we consider it *confirmed*.
- If the app contains a statement protected by the corresponding automatically granted permission, but our analysis does not consider it as reachable, we mark it as *unconfirmed*, as we want to be conservative in identifying protected accesses. Nevertheless, those statements might still be reachable, for instance, via inter-component communication.
- If the analysis does not find any statement protected by the permission under analysis, we mark the permission as *unused*.

4.2 Results

The results of the static analysis are summarized in Figure 6, where we report for each automatically granted permission the amount of *confirmed*, *unconfirmed* and *unused* instances. Since we do not see a significant difference in the two datasets for this analysis, we present the aggregated information from the ApkPure and Androzoo datasets.

The results clearly show that in more than half of the cases (lighter color of the bars) permissions are actually used by the application. Some permissions, such as reading SMS (75%) and phone numbers (84%), or accessing the precise location (74%), have much higher values. Additionally, write calendar (82%) and voicemail (100%) permissions have a very high confirmed frequency, however their low number of occurrences—less than 0,2% of the total combined—makes them less interesting for our analysis.

[8] https://developer.android.com/docs
[9] https://github.com/aosp-mirror

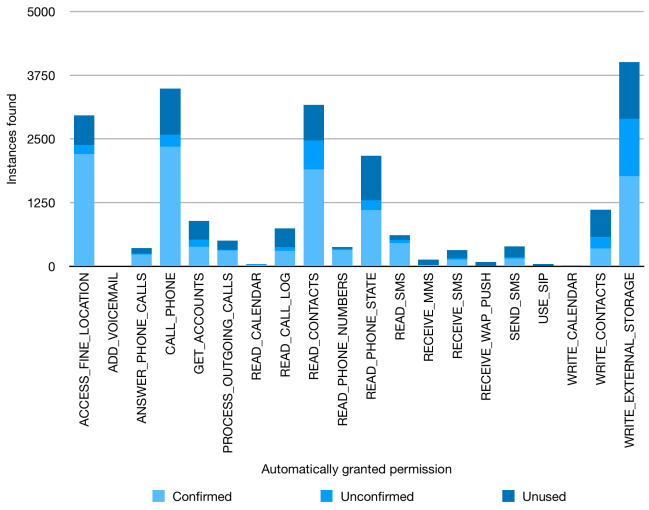

Figure 6: Automatically granted permissions in AndroZoo and ApkPure combined

We want to stress that these numbers are conservative, as our static analyses are not fully sound in the construction of the call graph, may ignore relevant permission-protected invocations, and ignore native code.

We verify if the application uses each of the automatically granted permissions and then focus our attention on *how* the app exploits it. More precisely, we want to understand whether apps access sensitive information and leak it over the Internet without the user noticing it.

We confirm that apps actually use at least 56% of the automatically granted permissions. Accurate user's location, phone numbers and SMS are the most used, with a frequency of at least 70% of the cases.

5 MANUAL ANALYSIS AND MAIN FINDINGS

Static analyses show that most apps actually use the newly granted permission at least once in the code. To conclude our study we thus thoroughly analyze the behavior of some of the apps that use automatically granted permissions.

To this end, we perform a dynamic analysis of a subset of these apps, manually running the app and profiling it to understand its behavior, and finally manually inspecting the bytecode if needed. We discuss how we perform the analysis in Section 5.1, and we present the most interesting findings in Section 5.2.

5.1 Dynamic Analyses and Manual Inspection

To understand the dynamic behavior of an app, we instrument it before exploration. We use Soot to insert logging code before each Android API invocation covered by permissions we aim to investigate. Beside logging specific method invocations, we record specific method arguments for `ContentResolver.query()`, `insert()` and `update()` method invocations, so that we log the data source URI and the query string that static analysis sometimes cannot resolve.

We optionally log particular variable values (for instance, to spot encrypted values) in order to identify a possible data leak.

We install the instrumented app on the device and manually use it extensively—aiming to exercise as many activities/behavior as possible. We do not resort to automated tools to explore the behavior of the app because we want to achieve a high behavior coverage, which is still a challenge for the state of the art tools [15].

To assess if an app leaks sensitive data protected by automatically granted permissions, we intercept its network traffic through a proxy. We resort to Charles, an HTTP proxy that allows analysts to intercept HTTP and HTTPS traffic between a device and the Internet[10]. Charles works out of the box for Android devices. After obtaining the traffic dumps captured by Charles, we manually analyze them, looking for the information protected by the automatically granted permission (e.g., IMEI of the phone in case an application has the READ_PHONE_STATE permission automatically granted). We do this type of analysis for permissions that read sensitive data from the phone, such as READ_CONTACTS or READ_CALL_LOGS. This analysis is far from being sound. First and foremost, we do not exhaustively explore all the app with our manual interactions. Secondly, apps often implement checks (e.g. SSL pinning) to avoid sending sensitive information through non-trusted proxies. In these cases we would not detect any information leak.

Many of the apps we consider in our analysis require valid credentials to work: we generate new accounts whenever possible. For the remaining cases—for example, banking apps, which require valid accounts we cannot generate—we only log behavior and network traffic generated from the screen we have access to, such as home and settings. Given the manual effort, we manually execute only 400 binary files, and we manually inspect only tens of them, selecting the most suspicious cases.

5.2 Confirmed Data Leaks

Due to space limitations we report the most interesting findings of our thorough analysis.

We observe that at least 12% of the apps leak the exact location of the user that they obtain through the automatically granted ACCESS_FINE_LOCATION permission. One app leaks the IMEI code of the phone, obtained thanks to the automatically granted READ_PHONE_STATE permission. The network traffic shows that these apps send exact location and IMEI mostly to what seem to be their subdomains (64% of the domains), but they also send this information to what seem to be third parties and mostly advertisement companies (36% of the domains). Here are some of these apps:

cn.xender Xender is a popular file and music sharing app with over 100M installs. It leaks the device serial number to four different urls: ac.mobileanapp, api.cloudmobi.new, aws.xenderbox.com, and fb.xanderbox.com.

fi.iltalehti.iltalehti Iltalehti is a Finnish news application with over 1M installs. It leaks the location to mediation.adnxs.com, app.iltalehti.fi, and m.iltalehti.fi.

air.com.interactech.moovz Moovz is a popular LGBT social network. It has over 1M installs and leaks the precise location to mobileapi.moovz.com.

cz.thran.flowerchecker FlowerChecker provides a plant identification service. It has over 100k installs and leaks the location to api2.flowerchecker.com.

me.nextplus.smsfreetext.phonecalls Nextplus is an app that provides free phone calls and messages. It has over 5M installs and leaks the location to ces.app.nextplus.me.

pr.nip.cennoticias Central das Noticias is a Portuguese free news app with over one million downloads. It leaks the location to ads.cennoticias.com and logging.cennoticias.com.

uzsusd.dollaruz.dollaruzbekistan DollarUz is an Uzbekistan finance and currency exchange calculator. It has 50k installations and leaks the location to onesignal.com.

vdm.activities VDM Officiel is a French comics app with 1M downloads. It leaks the location to ads.mopub.com.

We further analyze the descriptions of these apps. The authors of the paper agree that none of the apps that leak data would need the information that they collect: the app that leaks the IMEI could simply use the Android ID, which is not permission protected, to uniquely identify the device. Moreover, none of the applications leaking the location really needs access to the fine location of the user. These apps, which range from currency exchange calculators and social network to news applications or comics, could perfectly work by just knowing the user's coarse location.

Although the behavior of these apps may not look legitimate, they all state in their privacy policies that they collect the data being leaked, showing the intention/awareness of the developers, but at the same time protecting themselves from explicit privacy violations.

We now describe more in details the most suspicious apps we analyzed thoroughly:

com.cootek.smartinputv5 This popular app, with over 100M installs, adds a new smart keyboard for users to efficiently provide inputs. Version 5051 requests the permission to access the phone's *call logs*, effectively gaining access to the phone call history without any user's approval, and sending call log information to their company servers. The READ_CALL_LOG permission is granted automatically since in previous releases the app already requested the READ_PHONE_STATE permission, which lets the app read the phone state to access information such as the phone number of the device for unique identification, or to perform an action (e.g. close the keyboard) when a call is being received.

Profiling the execution, we *confirm* that the app accesses the list of phone calls listed on the device and leaks this information upon starting a new Activity to a server with a Chinese domain. This Activity, however, has to be explicitly invoked with an intent, since it is not reachable from the main Activity. Despite being hard to trigger, this behavior is highly suspicious as it effectively steals sensitive information that users did not grant permissions to read.

net.devmain.callblock "Call Block — number blacklist", according to its description, blocks unwanted incoming or outgoing calls with the support of a variety of filters. The app requests the

[10] https://www.charlesproxy.com/

READ_CONTACTS permission since its early releases, as this information is intuitively necessary for the app to function. Release 62, however, requests in the manifest the GET_ACCOUNTS permission, and the OS automatically grants it.
This release *effectively collects e-mails of specific domains* (filtered by gmail and hotmail keywords) from the accounts data on the device, ciphers them, and sends this information to their servers. Almost all strings in the app (including URLs) are encrypted with DES algorithm, which—along with obfuscation—hampers the manual inspection.
The information sent in the JSON request could be an authentication request: it contains *auth_key*, *auth_token* fields, and the data message with *username* and *password* keys. The value for *username* is always 'net.devmain.callblock'—the package name of the app, while *password* contains the ciphered e-mail in Base64 format. The data is sent to http://zxc-atari.rhcloud.com/rest/resource/zxcreg, which is no longer reachable.
The app does this operation *as a background task*, triggered from the main screen, and continues until it gets a response of successful data transmission. While this operation runs in the background, there is NO exposed functionality of the app that could justify this behavior (e.g., authentication). In the User Consent data section of Terms and Conditions, published on their web-site, the developers state they "use cookies and device data to personalize content and ads". E-mails may simply be used as identifiers for further in-app purchases, though the manners used for their collection and transmission make the activity quite suspicious.

com.adstick.goodhabit1 "Good Habit" is a quiz game for children, intended to teach good manners. Version 1.3, published on 9 August 2016, requests the GET_ACCOUNTS permission. Version 1.5, published only two days after (on 11 August), requests the READ_CONTACTS permission. Right after booting, the app retrieves all the entries in the contact list on the phone and sends them to http://www.kotadiya.com:3004/. At the time of the analysis this domain is no longer reachable.
The manual inspection of the bytecode confirms that in the `onCreate()` life-cycle method of the main activity the app establishes a socket connection with the server, checks if the contacts have been already uploaded, and then creates a separate background thread which queries all contacts data using the ContentResolver, storing it in a field. Afterwards, it bundles the contact data together with the IMEI number and sends the package via socket connection to the remote server. Meanwhile, the foreground Activity of the app suggests to log-in with a Facebook account. The app's description does not justify this behavior in any way, and the binary code clearly shows that the app steals user private data without any consent. This application has been removed from the Google Play store.

com.siftr.whatsappcleaner "Magic Cleaner for WhatsApp" has over 50K downloads. It uses deep learning to identify and delete the photos to trash in the WhatsApp folder. Version v1.5.2 requires the GET_ACCOUNTS permission, whereas release v1.6.1 also requests the READ_CONTACTS permission. With this latter automatically granted permission, the app collects contacts and sends them to the remote server, a behavior that cannot be justified for the application's functionality. Our manual inspection reveals that upon startup the app checks if the list of contacts has been already uploaded. If this is not the case, it queries the contact list and sends it along with the Android ID of the device to the developer's server https://siftr.co/whatsapp/60n2a62s/. Data acquisition and dispatch are performed by separate objects; the uploadContacts() method is invoked via a callback, which makes it hard for static analysis to detect. In the most recent releases of this app the leak no longer happens.

Despite showing some real cases where apps exploit permission groups to gain access to the user's sensitive information, we want to stress the fact that our analysis might have missed other occurrences of this behavior. Our findings, though, prove that the risk of granting permissions at group level is real and can be abused by developers.

Our analysis shows that some applications actually abuse automatically granted permission, leaking sensitive data without any user's authorization.

6 RELATED WORK

Many research papers study the evolution of Android applications [26, 29, 45]. Wei et al. [44] consider permission patterns and their distribution and report that applications tend to request more permissions over time and to become overprivileged. Calciati et al. [13, 14] confirm their results on a much larger dataset of 14,000 APKs. Moreover, they show that applications tend to slowly increase the number of sensitive information flows over time. Besides, apps use more API methods protected by dangerous permissions that were already granted. This insight motivated us to investigate the problem of automatically granted permissions in general. Wang et al. [43] perform a large-scale empirical study analyzing applications removed from the Google Play store between two snapshots taken in 2015 and 2017. It finds out that nearly half of them were removed, and over 20% of removed apps belong to 1% of the developers.

Other papers focus on the evolution of Android libraries: Book et al. [11] study the behavioral evolution of Android advertisement libraries, discovering that they increasingly take advantage of the app's permissions to gain access to private user data. Derr et al. [16] focus on libraries updatability and study the root causes of why Android app developers do not adopt new library releases. He et al. [25] perform an empirical study on evolution-induced compatibility issues in Android applications: consecutive Android releases can add a high number of changes that apps have to take in consideration. The study reports that three quarters of newly added methods introduced with a new Android release are not supported by the Android Support library, and that over 90% of the apps need to address compatibility issues introduced by new Android releases.

The research community analyzed how app developers look for alternative ways to gain access to permission-protected user data. Reardon et al. [33] implement a test environment to instrument and analyze apps behavior and network traffic, uncovering a number of side and covert channels currently in use by hundreds of popular apps which allows them to gain unauthorized access to permission-protected sensitive user data. Sadeghi et al. [36] build up on previous research [9, 12, 18, 46] of permission-induced security attacks—security breaches enabled by a permission misuse—and propose a permission analysis and enforcement framework that

considers the temporal aspects of permission-induced attacks for their detection and prevention. These studies are similar to ours in the sense that they study applications that leak sensitive user data. Nevertheless, while these studies focus on bugs or limitations in Android to circumvent the permission system, in our work we focus our analysis on apps that use a problem in the design of the permission model in Android.

More literature focuses on different aspects of the Android permission system [20–22, 37, 41]. Taylor and Martinovic [40] analyze the evolution of dangerous permissions by performing a broad study on 1.6M applications with Google Play Store snapshots taken quarterly over a year period. Stevens et al. [39] gathered insights about the discussion of permissions in online fora and their misuse, showing a direct relation between the misuse and popularity of the permission. Finally, Smullen et al. [38] study the actual privacy preferences of almost 1000 users toward three sensitive Android app permissions (calendar, location, contacts) and train a model for privacy preference recommendation combining both supervised and unsupervised learning.

Other studies focus on analyzing user's privacy. Feal et al. [19] study privacy issues, and as well the behavior evolution, of parental control apps. They identify several worrisome issues, such as personal information collection, lack of encryption in data transmission, and concerning privacy policies. Gamba et al [24], instead, present a large scale study of pre-installed software on Android devices from more than 200 vendors. They identify user tracking activities by pre-installed Android software as well as by embedded third-party libraries.

Ren et al. [34] perform a network traffic analysis on 7,665 versions of 512 apps to analyze what personally identifiable information appears in the traffic generated by those apps. Their study shows that privacy has worsened over time, and the information gathered by an application can change with different releases, limiting the validity of studies that focus on a single version of an app. Similarly Razaghpanah et al. [32] focus on third-party services whose main function relies on collecting tracking information from users (advertising and tracking services). They analyze the network traffic generated by apps to understand where sensitive data ends up, both considering parent companies of ADS, which can later combine and/or sell data to other companies, and how data flows across borders, with the corresponding impact of privacy regulations. Nguyen et al. [31] present a study relating app reviews with security and privacy related changes. They show that apps with a runtime permission handling receive a higher number of security and privacy reviews. The authors also could correlate almost half of the privacy-related reviews to third-party library code, showing that in most cases the reviews complained about a behavior introduced by the third-party code. Finally, other researchers analyze the user interface of Android apps to identify covert behavior [6, 27, 28, 35]. These contributions are related to our work, but do not specifically mention the issue of automatically granted permissions.

7 LIMITATIONS AND THREATS TO VALIDITY

A threat to the validity of our work is that our datasets may not be representative of the entire ecosystem: the majority of apps we analyzed were taken from the Google Play Store, which has been shown to be largely trustworthy [1, 30, 47]. We limit this risk by having in our dataset all the apps listed in ApkPure, an alternative app store for Android.

A limitation we have is that both our static and dynamic analyses might produce false negatives by failing to detect uses of automatically granted permissions. This is why we consider the number of leaks found as a lower bound. Static analysis may suffer from false negatives because the construction of the call graph is not sound, we do not analyze the native code, and the permission mapping we used may be incomplete. Dynamic analysis have false negatives by definition. By manually exercising the app we cannot be sure that we explore the whole behavior of the application. Moreover, apps have non deterministic behavior that we may miss. Finally, we used a proxy to intercept network traffic even when encrypted. Our approach would not recognize most of the data leaks in case they were encrypted or tampered by the application before being sent to the server. Moreover, Android apps often use SSL pinning to communicate sensitive information only to trusted servers (thus not our proxy). In all these cases we miss the information.

Finally, even when we did find applications leaking sensitive data, we did not further analyze the leaks from a network point of view to answer questions such as: what are the servers to which data is sent, and to which companies they belong to, which could provide additional insights on the app behavior.

8 CONCLUSIONS

In this paper we present an empirical study on 2,865,553 Android apps evaluating the threat posed by permission groups to the privacy and security of final users.

Our study shows that often apps request new dangerous permissions, and the Android OS automatically grants them. Our analyses show that apps actually use over 56% of the automatically granted permissions.

Finally, our manual inspection reveals clear abuses of apps that exploit automatically granted permissions to access sensitive user data—such as the accurate location or the list of contacts—and leak them either to servers belonging to the company developing the app or to third parties.

Several studies report the many flaws of the permission model implemented in Android. Our study highlights a concrete new threat. We are aware that balancing usability and privacy in mobile devices is a challenging task, and we acknowledge the effort that Android engineers are putting in enhancing the permission model by introducing new policies and fine grained permission groups, as highlighted in Section 2. However, as our study shows, this is not enough. User alerts, displayed only when the updated app uses the automatically granted permission, could enforce the privacy.

Find more information on the project at:

https://github.com/gorla/appmining

ACKNOWLEDGMENTS

This work was supported by the Spanish Government through the SCUM grant RTI2018-102043-B-I00 and the project DEDETIS, and by the Madrid Regional projects N-Greens Software (n. S2013/ICE-2731), BLOQUES and MadridFlightOnChip.

REFERENCES

[1] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. 2016. Empirical assessment of machine learning-based malware detectors for Android - Measuring the gap between in-the-lab and in-the-wild validation scenarios. *EMSE* 21, 1 (2016), 183–211.

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *MSR 2016: 13th Working Conference on Mining Software Repositories*. 468–471.

[3] Panagiotis Andriotis, Angela Sasse, and Gianluca Stringhini. 2016. Permissions Snapshots: Assessing Users' Adaptation to the Android Runtime Permission Model. In *WIFS 2016: Proceedings of the 8th IEEE Workshop on Information Forensics and Security*. 1–6.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI 2014: Proceedings of the ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation*. 259–269.

[5] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: analyzing the Android permission specification. In *CCS 2012: Proceedings of the 19th ACM Conference on Computer and Communications Security*. 217–228.

[6] Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, and Andreas Zeller. 2017. Detecting Behavior Anomalies in Graphical User Interfaces. In *ICSE 2017: Proceedings of the 39th International Conference on Software Engineering Companion*. 201–203. https://doi.org/10.1109/ICSE-C.2017.130

[7] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *CCS 2016: Proceedings of the 23rd ACM Conference on Computer and Communications Security*. 356–367.

[8] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *USENIX Security: 25th USENIX Security Symposium*. 1101–1118.

[9] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. 2015. Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification. In *FM 2015: 20th International Symposium on Formal Methods*. 73–89.

[10] Denis Bogdanas. 2017. DPerm: Assisting the Migration of Android Apps to Runtime Permissions. *CoRR* abs/1706.05042 (2017). http://arxiv.org/abs/1706.05042

[11] Theodore Book, Adam Pridgen, and Dan S. Wallach. 2013. Longitudinal Analysis of Android Ad Library Permissions. *CoRR* abs/1303.0857 (2013).

[12] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2012. Towards Taming Privilege-Escalation Attacks on Android. In *NDSS 2012: 19th Annual Symposium on Network and Distributed System Security*.

[13] Paolo Calciati and Alessandra Gorla. 2017. How Do Apps Evolve in Their Permission Requests? A Preliminary Study. In *MSR 2017: 14th International Conference on Mining Software Repositories*. 37–41.

[14] Paolo Calciati, Konstantin Kuznetsov, Bai Xue, and Alessandra Gorla. 2018. What did Really Change with the new Release of the App?. In *MSR 2018: 15th International Conference on Mining Software Repositories*. 142–152.

[15] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*. IEEE Computer Society, 429–440. https://doi.org/10.1109/ASE.2015.89

[16] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *CCS 2017: Proceedings of the 24th ACM Conference on Computer and Communications Security*. 2187–2200.

[17] Daniel Domínguez-Álvarez and Alessandra Gorla. 2019. Release Practices for iOS and Android Apps. In *WAMA 2019: Proceedings of the 4nd International Workshop on App Market Analytics*. 15–18.

[18] Zheran Fang, Weili Han, and Yingjiu Li. 2014. Permission Based Android Security: Issues and Countermeasures. *Computers & Security* 43 (06 2014), 205–218.

[19] Alvaro Feal, Paolo Calciati, Narseo Vallina-Rodriguez, Carmela Troncoso, and Alessandra Gorla. 2020. Angel or Devil? A Privacy Study of Mobile Parental Control Apps. In *The 20th Privacy Enhancing Technologies Symposium (PoPETs 2020.2)*. 314—-335.

[20] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *CCS 2011: Proceedings of the 18th ACM Conference on Computer and Communications Security*. 627–638.

[21] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David Wagner. 2012. How to Ask for Permission. In *USENIX HotSec 2012: Proceedings of the 7th USENIX Workshop on Hot Topics in Security*.

[22] Adrienne Porter Felt, Serge Egelman, and David Wagner. 2012. I've Got 99 Problems, but Vibration Ain'T One: A Survey of Smartphone Users' Concerns. In *SPSM 2012: Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. 33–44.

[23] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *SOUPS 2012: Proceedings of the Eighth Symposium on Usable Privacy and Security*. 1–14.

[24] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. 2020. An Analysis of Pre-installed Android Software. In *IEEE S&P: 2020 IEEE Symposium on Security and Privacy*.

[25] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and Detecting Evolution-induced Compatibility Issues in Android Apps. In *ASE 2018: Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*. 167–177.

[26] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Tracking the Software Quality of Android Applications Along Their Evolution. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*. Washington, DC, USA, 236–247.

[27] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *ICSE 2014: Proceedings of the 36th International Conference on Software Engineering*. 1036–1046.

[28] Konstantin Kuznetsov, Vitalii Avdiienko, Alessandra Gorla, and Andreas Zeller. 2018. Analyzing the User Interface of Android Apps. In *MobileSoft 2018: Proceedings of the 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. 84–87.

[29] Maleknaz Nayebi, Konstantin Kuznetsov, Paul Chen, Andreas Zeller, and Guenther Ruhe. 2018. Anatomy of Functionality Deletion: An Exploratory Study on Mobile Apps. In *MSR 2018: 15th International Conference on Mining Software Repositories*. 243–253.

[30] Yi Ying Ng, Hucheng Zhou, Zhiyuan Ji, Huan Luo, and Yuan Dong. 2014. Which Android App Store Can Be Trusted in China?. In *COMPSAC 2014: Proceedings of thehe 38th Annual International Computers, Software & Applications Conference*. 509–518.

[31] Duc Cuong Nguyen, Erik Derr, Michael Backes, and Sven Bugiel. 2019. Short Text, Large Effect: Measuring the Impact of User Reviews on Android App Security & Privacy. In *IEEE S&P: 2019 IEEE Symposium on Security and Privacy*.

[32] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. 2018. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *NDSS 2018: 25th Annual Symposium on Network and Distributed System Security*.

[33] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *USENIX Security: 28th USENIX Security Symposium*. 603–620.

[34] Jingjing Ren, Martina Lindorfer, Daniel J. Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. 2018. Bug Fixes, Improvements, ... and Privacy Leaks. In *NDSS 2018: 25th Annual Symposium on Network and Distributed System Security*.

[35] Julia Rubin, Michael I. Gordon, Nguyen Nguyen, and Martin Rinard. [n.d.]. Covert Communication in Mobile Applications. In *ASE2015*. 647–657.

[36] Alireza Sadeghi, Reyhaneh Jabbarvand, Negar Ghorbani, Hamid Bagheri, and Sam Malek. 2018. A Temporal Permission Analysis and Enforcement Framework for Android. In *ICSE 2018: Proceedings of the 40th International Conference on Software Engineering*. 846–857.

[37] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2017. PATDroid: Permission-aware GUI Testing of Android. In *ESEC/FSE 2017: The 25th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 220–232.

[38] Daniel Smullen, Yuanyuan Feng, Shikun Aerin Zhang, and Norman Sadeh. 2020. The Best of Both Worlds: Mitigating Trade-offs Between Accuracy and User Burden in Capturing Mobile App Privacy Preferences. *PETS* 2020, 1 (2020), 195–215.

[39] Ryan Stevens, Jonathan Ganz, Vladimir Filkov, Premkumar Devanbu, and Hao Chen. 2013. Asking for (and About) Permissions Used by Android Apps. In *MSR 2013: 10th Working Conference on Mining Software Repositories*. 31–40.

[40] Vincent F. Taylor and Ivan Martinovic. 2017. To Update or Not to Update: Insights From a Two-Year Study of Android App Evolution. In *ASIACCS 2017: Proceedings of the ACM Asia Conference on Computer and Communications Security*. 45–57.

[41] Guliz Seray Tuncay, Soteris Demetriou, Karan Ganju, and Carl A. Gunter. 2018. Resolving the Predicamentof Android Custom Permissions. In *NDSS 2018: 25th Annual Symposium on Network and Distributed System Security*.

[42] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot – a Java Bytecode Optimization Framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. 13–23.

[43] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. 2018. Why Are Android Apps Removed from Google Play?: A Large-scale Empirical Study. In *MSR 2018: 15th International Conference on Mining Software Repositories*. 231–242.

[44] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012. Permission Evolution in the Android Ecosystem. In *ACSAC 2012: Proceedings of the 28th Annual Computer Security Applications Conference*. 31–40.

[45] Jack Zhang, Shikhar Sagar, and Emad Shihab. 2013. The Evolution of Mobile Apps: An Exploratory Study. In *DeMobile 2013: 1st international Workshop on Software Development Lifecycle for Mobile*. 1–8.

[46] Yajin Zhou and Xuxian Jiang. 2013. Detecting Passive Content Leaks and Pollution in Android Applications. In *NDSS 2013: 20th Annual Symposium on Network and Distributed System Security*.

[47] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *NDSS 2012: 19th Annual Symposium on Network and Distributed System Security*.