Investigation on the Android Permission System

Zhijie Xia
University of Calgary
zhijie.xia@ucalgary.ca

Joel Reardon
University of Calgary
joel.reardon@ucalgary.ca

***Abstract*—Smartphones have bought great convenience to our everyday life. Among many smartphone operating systems, Android is one of the most used operating systems. Android uses a permission system to ensure security and protect user privacy. Unfortunately, many severe attacks and privilege escalations have been found on the Android platform. While previous efforts have achieved success in detecting malicious apps on Android, they failed to thoroughly examine the design and implementation of the Android permission system, which enables such malicious behavior. In this paper, we inspected the design and implementation of the permission system in detail. We performed static analysis on the Android source and extracted 1,976 permission-guarded APIs from the Android framework. In addition, we reverse-engineered 35, 117 third-party apps from Google Play Store to understand the framework API usage of real-world applications.**

Our research shows that Android permissions are applied unevenly to protect APIs, with as many as 159 being guarded and as few as one. Nearly 99% of apps request access to the Internet, even those labeled as offline. Additionally, `AlarmManager` methods are used by apps to periodically check the user's location even when the app is not in use. Our research highlights the limitations of the current Android permission system and underscores the need for further investigation to enhance its design and implementation.

I. Introduction

The Android Open Source Project, with the support of Google and the open-source community, has experienced tremendous growth in recent years. Industry reports [1], [2] indicate that Android has captured an overwhelming 71 percent of the smartphone market share and more than half of the tablet market share, with a massive 2.7 billion active users worldwide.

This massive user base, combined with Android's market dominance, makes the platform a prime target for malicious actors seeking to exploit sensitive information, such as contacts, SMS messages, photos, and other sensitive data stored [3], [4] on the device. Additionally, the open-source nature of Android makes it more vulnerable to attacks and exploitation as adversaries have access to the source code.

Given the vast scale and widespread popularity of the Android platform, it is critical to prioritize securing and protecting the privacy of the platform and its users. However, previous research has mainly focused on finding malicious apps [5]–[7] and discovering permission issues in third-party [8]–[10] and preinstalled apps [11], [12]. Although several studies have monitored the evolution [13]–[15] of the Android permission system, their findings are either outdated or insufficient to provide insights into the effectiveness of the Android permission system. We refer interested readers to Section III for the previous work.

In this paper, we take steps toward investigating the design and implementation of the Android permission system to provide a better understanding of the effectiveness of the current Android permission system. First, we scanned through the Android source and built the most updated API-permission mapping [16], [17], which consists of 1,976 API calls and 323 permissions. Second, we crawled and reverse-engineered 35, 117 third-party apps from Google Play Store to build a database of real-life API call instances. Lastly, by combining the API-permission mapping and the real-life API call instances, we examined the usage of the permission-guarded APIs using a frequent itemset mining algorithm - FP growth algorithm [9], [18].

Additionally, we performed a detailed case study and evaluated the security implications for our static analysis approach. The result confirmed the significance of our findings on the current Android permission system.

In summary, our contributions are the following

- We performed an up-to-date systematic study on Android permission system. The empirical results from our study can guide future research and offer valuable understanding of the Android permission at API-level of details.
- We built the most up-to-date API-permission mapping. We found an uneven distribution of permission usage in Android framework. While three permissions are widely used to guard more than 100 APIs, there are 88 permissions are only used to guard a single API.
- We sampled 1,976 third-party apps and analyzed their manifest files. We discovered over 99 percent of our samples requested Internet access in their manifest files, while 3,989 of them listed **offline** as a keyword in their descriptions.
- We decompiled our app corpus and processed the real-world instances of permission-guarded APIs into the FP Growth algorithm to extract common usage patterns. We revealed the suspicious intimacy between `AlarmManager` and `LocationManager`.
- We randomly sampled 20 apps from app corpus and conducted detailed case studies. We found evidence supporting the use of the FP growth algorithm to understand API usage patterns. Specificially, We discovered a alarming coupling between `AlarmManager` and `LocationManager` in one of the sampled 20 apps.

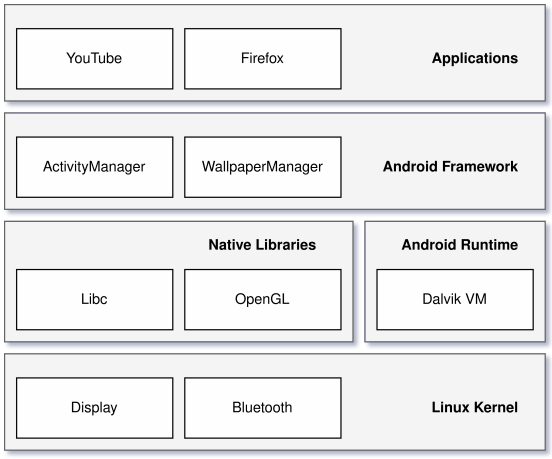

Fig. 1. The architecture of Android Platform

We discussed the potential security implications for such coupling and the innovative attacks that the coupling could enable.

- We discovered descrpiencies between the permissions requested in the manifest files and the inferred permissions from API call instances. We observed that these inferred permissions were more privileged, and we attribute this to the opportunistic data-stealing behaviors of third-party SDKs. These SDKs often contain contain deprecated code and lack maintenance, leading to the invocation of highly privileged operations.

The rest of this paper is organized as follows: Section II provides background on Android OS. Experienced readers may start at Section III for related work. Section IV and Section V describes the workflow, implementation, and results of our study. In Section VI, we validate our approach with a detailed case study. Limitations of our current implementation and future research opportunties are discussed in Section VII. Finally, this paper is concluded in Section VIII.

II. Background

A. Android Permission Level

Android uses a permission-based access control system to ensure users' security and privacy based on the security principle of least privilege. There are three types of permissions in the Android system described in the Android Dev Document [19], *install-time* permissions, *runtime* permissions and *special* permissions [1]. *Install-time* permissions are granted when the user installs an application, and a notice is presented to the user to request permission. *Runtime* permissions give apps more access to sensitive information. A runtime prompt would be presented to the user before the OS allows the app to access private data. The device manufacturers define *special* permissions , and applications with special permissions can usually perform potent actions.

[1]*Install-time*, *runtime*, and *speical* permissions are also known as *normal*, *dangerous* and *signature* permissions respectively.

B. Android Permission Enforcement

The foundation of the Android platform is the Linux kernel [20], as shown in Figure 1. Android builds on top of the Linux kernel, therefore, it inherits many key security features from the Linux kernel, such as *process isolation*, *secure IPC*, and *user-based permission model*. Moreover, every application on Android executes under a distinct UID in the Linux kernel. Whenever an application attempts to access sensitive information or claim system resource, the Linux kernel will check the application to see if its associated UID hold the required permissions. Permissions in Android are represented in hardcoded strings, and application developers can request any permission by adding a *use-permission* entry like `<uses-permission android:name="android.permission.VIBRATE"/>` in the *AndroidManifest.xml* file located in the apk. However, adding arbitrary *use-permission* entries does not necessarily imply that the app will be granted those permissions because the user can decline the request in runtime or cancel the installation process.

C. Android Framework API

As shown in Figure 1, Android applications and the Android framework are located in level 1 and level 2 on the stack. The immediate connection between applications and the framework is intentionally designed [21] so that the Android framework can efficiently provide a rich set of features for the application developers to construct their apps. Whenever an application developer wishes to use a feature, the developer invokes one or more of the publicly available APIs the Android framework provides. Those APIs are written in Java; for example, *ActivityManager* has functions `moveTaskToFront` and `moveTaskToBack`. The `moveTaskToFront` allows application developers to move their apps into the front UI so the apps are visible to the users. Similarly, `moveTaskToBack` moves apps into the background and makes apps invisible to the users. Some Android framework APIs allow apps to access sensitive resources like contact information and current location. Thus, the framework APIs enforce permissions by programmatically querying the system to determine whether the calling app has the required permission.

III. Related Work

Our work is built on a broad literature in the field of API-permission mapping and third-party application analysis for Android. Unfortunately, while prior third-party application analyses [16], [17], [22] to permission mappings claimed their results will last for future Android versions, all those permission mappings are found to be outdated in the latest Android version 13 (API level 33). Previous studies on third-party application analysis have shown success in catching misbehaved apps [5]–[7], [23], [24]. However, those works could have explained why the Android permission system allowed those misbehaviors to exist in the first place. In contrast, our work combines both permission mapping and third-party application analysis to inspect the Android permission system

and intends to determine whether the Android permission system is well-designed and carefully implemented.

A. API-Permission Mapping

Felt et al. [16] introduced *Stowaway*, a static analysis tool that detects overprivileged third-party applications. They used feedback-directed testing and API fuzzing to observe the permissions required to interact with system APIs. As a result, *Stowaway* constructed the first API-permission mapping that associates each framework API to a specific permission.

Based on *Stowaway*, Au et al. [17] designed and implemented *PScout*. *PScout*, in contrast to *Stowaway*, performed a static reachability analysis between API calls and permission checks to produce a specification that lists all the permissions that every Android API call requires. The specification demonstrates a significant improvement in API coverage compare to *Stowaway*.

The most recent state-of-the-art API-permission mapping was built by Backes et al. [22] using their *Axplorer*. *Axplorer* performed a static reachability analysis similar to *PScout*; however, the reachability analysis was conducted at more carefully selected entry points. Their evaluation found *Axplorer* achieved a more precise mapping result compared to *PScout* and raised questions about the validity of some prior works [17].

B. Third-party Application Analysis

Reardon et al. [5] looked for evidence of side- and covert-channels in practice and determined how the unauthorized access occurred. They designed a pipeline to run *88,000* third-party apps in an instrumented environment to find apps that exploit side- and covert-channels. Furthermore, they reverse-engineered those misbehaving apps and discovered covert and side channels used in the wild that compromise users' location data and persistent identifiers.

Kim et al. [6] created *FraudDetective*, a dynamic testing framework that identifies ad fraud activities in Android apps. They evaluated *FraudDetective* on *48,172* apps from Google Play Store and successfully identified 34,453 ad fraud activities in 74 apps.

IV. Methodology

The overall workflow of our static analysis is as follows (also illustrated in Figure 2). We inspected a subset of Android source code under the `frameworks` directory and extracted methods guarded by at least one permission into the *permission mapping* database. Concurrently, we executed another script that crawls apps from Google Play Store and decompiled the apks using existing tools [25], [26] into the *API usage* database. After we obtianed both databases, we filtered out the instances in the *API usage* database that do not require permissions. Consequently, we gained a list of real-world APIs that requires at least one permission.

Furthermore, we processed the list into a frequent itemset mining algorithm - FP growth. Our intend is to observe permission-guarded API usage patterns in the hope of gaining insights into how commonly app developers use different APIs in their apps. The rest of this section discusses the detailed implementation of the workflow. We discuss the results of our study in Section V.

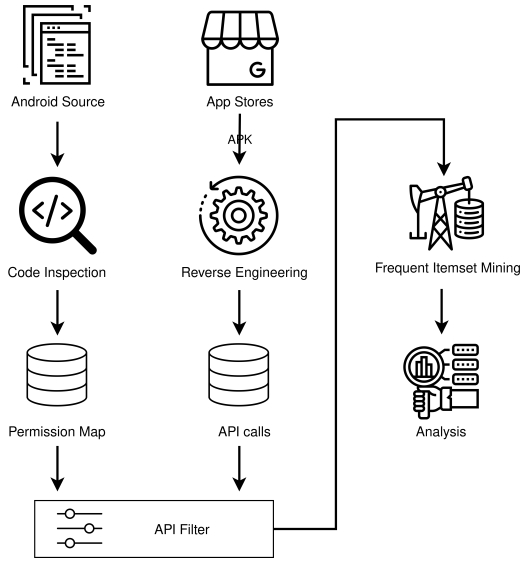

Fig. 2. Workflow of our static analysis.

A. API-permission Mapping

Although Android research has a long history on API-permission mapping, previous work [16], [17], [22] is antiquated, and their methodology and results only cover several API levels up to API 25. While the current Android functions on top of API 33, we have no choice but to build our permission map from scratch.

Java Annotation: As aforementioned, the Android framework is written in Java. Android developers have adopted the Java annotation to provide more comprehensive information in conjunction with code comments. Our construction of the permission map is based on the convention that whenever a developer writes a method requiring certain permissions to execute, the developer is obligated to add the `RequiresPermission` annotation on that particular method (see Listing 1).

```
@RequiresPermission(android.Manifest.permission.SET_TIME)
public void setTime(long millis) {
    try {
        mService.setTime(millis);
    } catch (RemoteException ex) {
        throw ex.rethrowFromSystemServer();
    }
}
```

Listing 1. An Android framework method with '@RequiresPermission' annotation

Permission API: By leveraging the `RequiresPermission` convention, we used *grep* to find the methods that are being guarded by permissions in the Android framework. Furthermore, we put each method against our crafted regular expression to extract a tuple that consists of

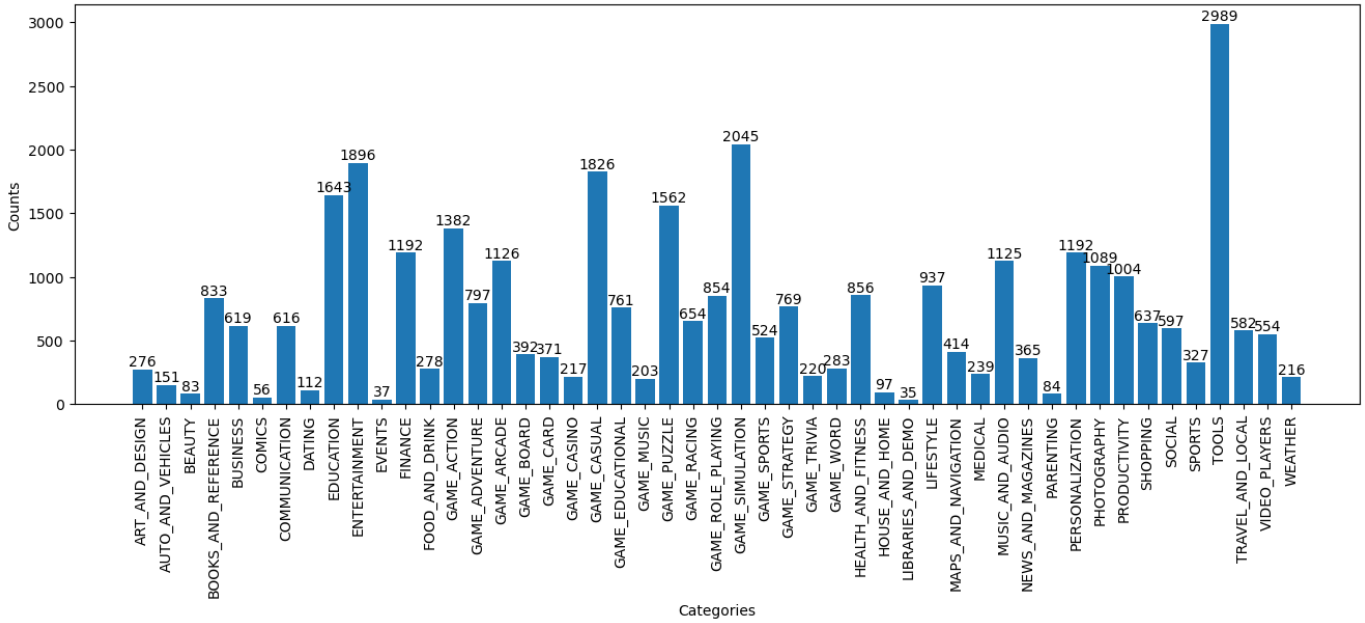

Fig. 3. Genre distribution of our App corpus

# of downloads	# of apps
∼100K	2,699
100K∼500K	6,372
500K∼1M	4,792
1M∼5M	12,595
5M∼10M	3,726
10M∼50M	3,988
50M∼	945
Total	35, 117

TABLE I
DISTRIBUTION OF THE NUMBER OF DOWNLOADS IN OUR CRAWLED APPS

the *file name*, *method name*, *required permission*, *parameter list*, and *return type*. We recorded the extracted tuples into a SQLite database and concluded our API-permission mapping.

B. Third-party Applications

In parallel with API-permission mapping, we performed a large-scale application collection and decompilation to obtain real-world instances of framework API usage.

App Collection: We wrote an apk crawler to download both apk and metadata from the Google Play Store. Since the Play Store has more than 2 million apps, it is infeasible for us to download every single one of the apps. Thus, we sampled 35, 117 apps according to app's number of downloads. Table I shows the distribution of the downloads on our app corpus. Furthermore, we parsed the category information for each app through the metadata. Our app collection comprises 48 categories, and their distribution is illustrated in Figure 3.

Decompile: Our goal is to find out the framework API usage. In order to do that, we utilized *apktool* [25] to decompile each apk into *smali* packages(an intermediate form of the app) and extracted java bytecode from each apk. Subsequently, we removed the bytecode that does not invoke framework APIs and store the rest of the bytecode alongside all the `AndroidManifest.xml` in the decompiled packages for further analysis.

C. Frequent Itemset Mining

After collecting 35, 117 apks and all API calls that they make, we used our permission map to exclude any API calls that are not protected by any permissions. The remaining API calls are further processed by a frequent itemset mining algorithm to identify patterns in the usage of the Android permission system by developers.

FP growth Algorithm: We further processed our data using the FP growth algorithm [18], a frequent itemset mining algorithm, to analyze our data to identify association rules and closely related items.

Since our items are permission-guarded API calls, this can reveal potential relationships, such as a correlation between two permission-guarded APIs, suggesting that if one is used, the other may also be invoked.

V. RESULTS

In this section, we delve into the results of our extensive static analysis, presenting a comprehensive overview of our findings and observations.

A. API-permission mapping

Our analysis of the Android framework source code extracted 1,976 APIs and their accompanying permissions. We

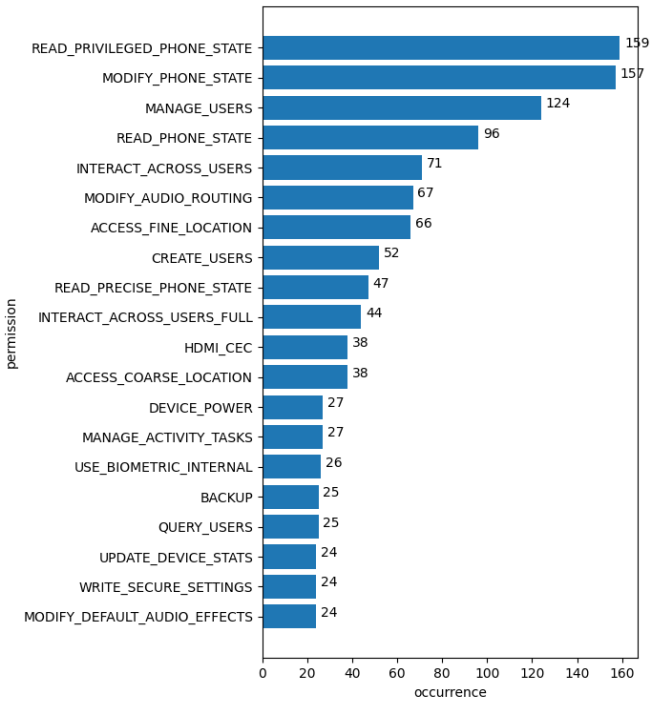

Fig. 4. Top 20 most used permissions to protect API calls

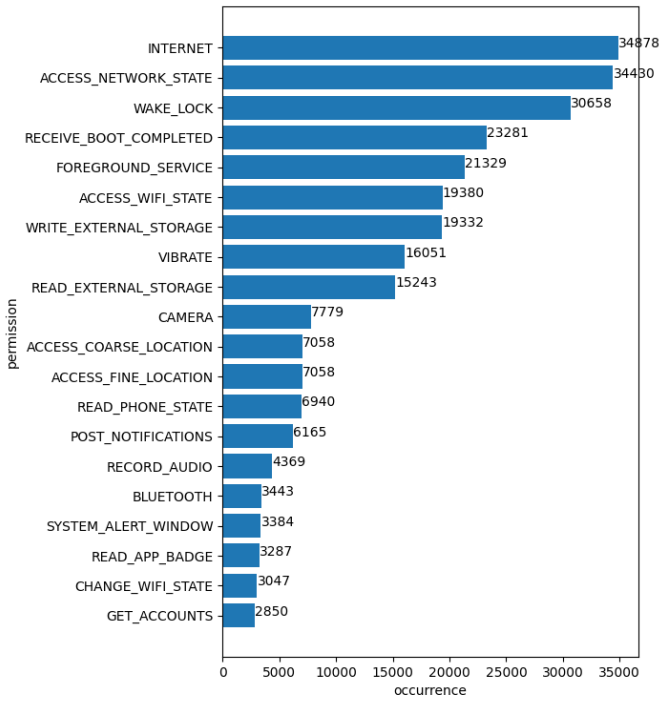

Fig. 5. Top 20 most requested permissions in collected manifests

found that the majority of permission-guarded APIs(1607) are secured by a single permission, while a considerable amount of APIs require multiple permissions to be able to executed. We then counted the number of APIs protected by each permission for each permission guards and identified the most used permissions in the framework. The top 20 most frequentlyused permissions for protecting framework APIs are presented in Figure 4.

A substantial subset of permission-guarded APIs are dedicated to provid features to manage the device states. In the top 20 most frequent used permissions, `READ_PRIVILEGED_PHONE_STATE`(159), ,`MODIFY_PHONE_STATE`(157), `READ_PHONE_STATE`(96) and `READ_PRECISE_PHONE_STATE`(47) account for 23.94% of the total extracted APIs, with each accounting for 8.3%, 8.2%, 5.0% and 2.5% respectively. `READ_PRIVILEGED_PHONE_STATE`, `READ_PHONE_STATE`, and `READ_PRECISE_PHONE_STATE` allow apps to attain vast information related to the current device states. `READ_PRIVILEGED_PHONE_STATE` and `READ_PHONE_STATE` are similar in that they both can allow apps to access phone state information such as device ID, phone number. However, `READ_PRIVILEGED_PHONE_STATE` allows access to more sensitive information than the `READ_PHONE_STATE`. For example, it allows access to information about the cell tower that the device is connected to and its signal strength, which is not available through `READ_PHONE_STATE`. It is worth noting that `READ_PRIVILEGED_PHONE_STATE` is signature-level permission and is intended be included in system apps and but not for third-party apps, while `READ_PHONE_STATE` is normal permission that is granted automatically to apps and do not require signature-level permission. Additionally, `READ_PRECISE_PHONE_STATE` permission provides access to extra information about the phone state, such as the battery level, signal strength, and network information. It requires signature-level permission and is intended for use by system apps and is not available to third-party apps. In contrast to the finite-granted read state permissions, there is only one permission called `MODIFY_PHONE_STATE` to modify the phone states. The `MODIFY_PHONE_STATE` allows apps to change state information on the device with a similar API method name.

Furthermore, we discovered that there are 88 permissions, each guarding a single API. While these permissions may pose a challenge for developers to memorize for specific use cases, they tend to protect the extremely critical resource. For instance,the *signature*-level permission `CLEAR_APP_USER_DATA` protects the `clearApplicationUserData` API. If an application holds `CLEAR_APP_USER_DATA` permission, it can clear all the data of another application, potentially resulting in significant data loss without the user's awareness.

B. Third-party Applications

We have collected 35, 117 apps into our app corpus. We further decompiled the apps and analyzed the Android framework API usage and permission requests.

Android Framework APIs and Features: We discovered that the number of API calls and the number of permission-guarded API calls vary largely across different files. The mean number of API calls was 46345.52with a standard derivation of 39332.44. Similarly, the mean number of permission-guarded API calls was 35.05 with a standard derivation of 29.63. These statistics suggest that most framework features can be provided without involving permissions.

Requested Permissions: As aforementioned, whenever an application needs a certain permission, the application developers must request the permission in the `AndroidManifest.xml`. We gathered 35, 117 manifest files and analyze `use-permission` entries to determine the most frequently requested permissions our app corpus. We list the top 20 most requested permissions in Figure 5.

At the very top of the list, we find that `INTERNET` permission is being requested by over 99% of apps. The unrestricted Internet access is alarming, as pointed out by previous studies [27]. We did a reverse lookup to the apps that request the `INTERNET` permission using simple string operations. We found a set of 3,989 apps put **offline** as a keyword in their descriptions while listing `INTERNET` as an entry in the `AndroidManifest.xml` in our app corpus. Based on previous literature [6], [7], [27], we hypothesize that many offline apps only need access to the Internet because developers use third-party SDKs to monetize their apps to display ads or use user devices as proxy nodes. However, unnecessary Internet access enlarges the attack surface and creates additional channels for attackers to exploit the system.

C. Invocation Permissions

By mapping each API-call to its associated permission using the mapping map that we attained from Section IV-A. We were able to identify the most needed permissions from the third-party applications. We lists the top 20 in the Figure 6. Among the 20 most used permissions implied by the API-calls, we find the android developers did not follow the least-privileged development principle, as evidently there are 11 permissions on the list are either system-level or dangerous-level permissions.

System-level Permission: Among the 20 permissions, 6 of them, `INTERACT_ACROSS_USERS`, `MANAGE_USERS`, `MODIFY_PHONE_STATE`, `READ_PRIVILEGED_PHONE_STATE`, `PACKAGE_USAGE_STATS`, `USE_BIOMETRIC_INTERNAL`, are system-level permissions and system-level permissions are often powerful and can pose serious security risks and privacy compromises if misused by malicious apps.

However, our app corpus contains primarily third-party applications and in general it can not be done to grant third party apps with system level permission, but there are some workarounds and exploits on rooted devices to grant applications with system-level permission. Again, letting third party apps to have system-level permissions are extremely dangerous and it is generally not adviced for third-party applications to attempt to gain system-level permissions.

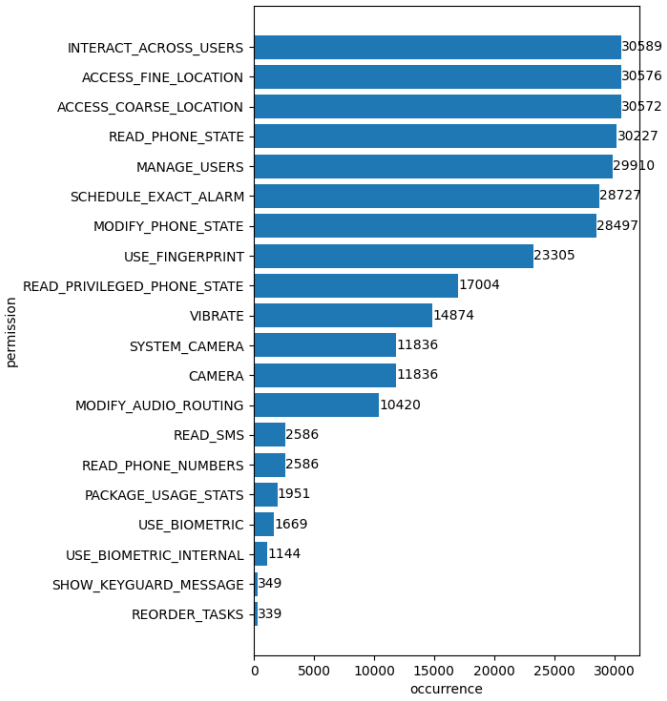

Fig. 6. Top 20 most associated permissions in all API instances

Dangerous-level permissions: Among the 20 most used permissions, we observed 5 dangerous-level permissions, `ACCESS_FINE_LOCATION`, `ACCESS_COARSE_LOCATION`, `CAMERA`, `READ_SMS`, `READ_PHONE_NUMBERS`. Dangerous permissions can give malicious applications with a ton of sensitive information if being granted. For example, the `ACCESS_FINE_LOCATION` can be used to track the user's real time location into radius of 5 meters errors. `READ_SMS` would allow an app to read SMS all messages stored in the device. It is adviced that third-party apps should avoid requesting dangerous-level permissions unless absolutely necessary for applications.

D. Frequent Itemset Mining

We run the `FP growth` algorithm on our permission-guarded API instances to find the correlation between different APIs. We discovered 100 sets of permission-guarded APIs width a minimal *support* of *0.5*. Within the 100 itemsets, there were 6 itemsets of size 1, and 94 itemsets of size greater than 1. The occurrences of size-1 itemset indicate high usages of the API in that itemset in the collected apps. For example, itemset `[getLastKnownLocation]` with a support of 0.8704 in Table II indicates that 87% of apps(30567 out of 35, 117) in the corpus used the function `getLastKnownLocation` in `LocationManager`. The occurrences of size-2 itemset suggests clustering of permission-guarded APIs , and the APIs in the same set can be grouped as they are frequently used together by developers. For instance, Support:0.7778, Itemset:[isUserUnlocked, getLastKnownLocation] in Table III suggests that there

Support	Itemset	Permissions
0.8704	[getLastKnownLocation]	['ACCESS_COARSE_LOCATION,ACCESS_FINE_LOCATION']
0.8509	[isUserUnlocked]	['MANAGE_USERS,INTERACT_ACROSS_USERS']
0.8180	[setExact]	['SCHEDULE_EXACT_ALARM']
0.8078	[requestAudioFocus]	['MODIFY_PHONE_STATE']
0.7980	[getNetworkType]	['READ_PHONE_STATE']
0.6636	[hasEnrolledFingerprints]	['USE_FINGERPRINT,INTERACT_ACROSS_USERS']
0.4751	[getDeviceId]	['READ_PRIVILEGED_PHONE_STATE']
0.4197	[vibrate]	['VIBRATE']
0.3813	[getImei]	['READ_PRIVILEGED_PHONE_STATE']
0.3445	[getCurrentLocation]	['ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION']
0.3370	[openCamera]	['SYSTEM_CAMERA, CAMERA']
0.3359	[getSubscriptionId]	['READ_PHONE_STATE']
0.3098	[getDataNetworkType]	['READ_PHONE_STATE']
0.2967	[requestAudioFocus]	['MODIFY_AUDIO_ROUTING','MODIFY_PHONE_STATE']

TABLE II
ITEMSETS SIZE OF 1

78% apps invoke the method `isUserUnlocker` in `UserManager` and the method `getLastKnownLocation` in `LocationManager` at the same time in apps.

Frequently used APIs: Our analysis uncovered 6 APIs that are extensively used in various apps, shown in Table III. It should be noted that `getLastKnownLocation` and `getNetworkType` have known security implications as apps that use `getLastKnownLocation` can gain the user's location, and `getNetworkType` can reveal if the user is connected to WiFi or Cellular, inferring whether the user is indoor or outdoor. The high usage of these APIs in multiple apps raises concerns and emphasizes the need for thorough evaluations of privacy and security risks before implementing APIs in app development.

AlarmManager: We noticed that 48 out of 100 association rules have a method from `AlarmManager`. The `AlarmManager` contains a handful set of methods that allow application developers to have their apps run at a specific time in the future, even if the apps are not currently running. By coupling `AlarmManager` with other APIs, apps can deliver scheduled services, such as displaying notifications at a particular time. Moreover, we found various itemsets that contain both method `set` from `AlarmManager` and method `getLastKnownLocation` from `LocationManager`. This intimacy between `AlarmManager` and `LocationManager` indicates that apps constantly query the user's location even when the apps are not running. Unfortunately, this raises a high privacy concern because we could not provide any justifications for such behavior except there is a widespread personal information collection ongoing on Android, as previously discussed by Shen et al. [28].

VI. CASE STUDY

In order to better understand the security implications of the current Android development, we conducted manual inspections of the reverse engineered smali packages. The inspections are aimed at providing insights and evidence that support our findings and approach. We randomly selected 20 apps for inspection and analysis, and the app ids are listed in Table IV. It took one researcher approximately 5 hours to complete the inspection.

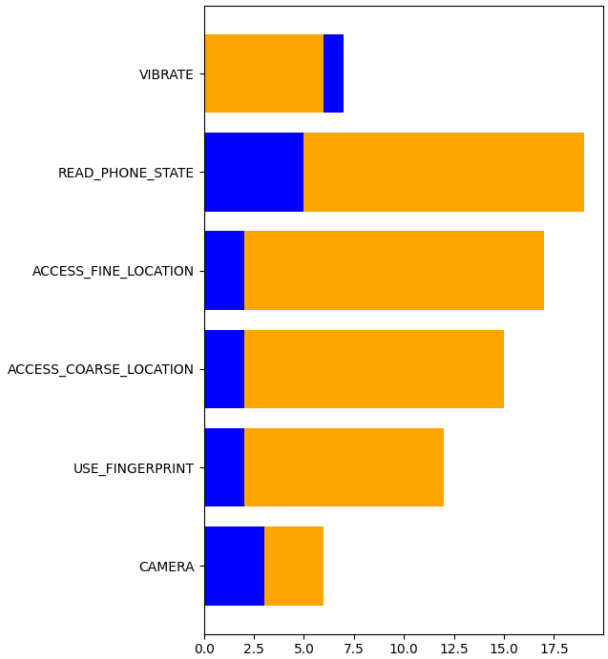

Fig. 7. Permissions requested in manifest(blue) vs permissions needed to execute the app

A. Discrepancies between Requested and Inferred Permissions

The observant reader might have noticed that in Figure 5 and Figure 6, the permissions requested in the manifest does not actually match the permissions inferred from the API call instances. In fact, there tend to more permissions are actually required from API call instances than the permissions requested in the manifest file. We observed similar discrepancies in our 20 app sample as illustrated in Figure 7. These discrepancies raise concerns potential security risks, as the app are might access more sensitive data without explicit user's consent.

We also made an observation that the actual requested permissions, as inferred from API call instances, tend to create greater security and privacy risks. For instance, `READ_PHONE_STATE` provides extensive device information that could lead to device fingerprinting [29] and enable malicious apps to detect emulator environment [30]. This raises the question of why there is a significant increase in the number of permissions inferred from API call instances over the the number of permissions requested in the manifest, given that an app cannot be granted with more permissions that it actually requested.

Our inspection revealed that Third-party SDKs are a major contributor to the increase in inferred app permissions. While developers generally follow least-privileged principle and re-

Support	Itemset	Permissions
0.7778	[isUserUnlocked, getLastKnownLocation]	['MANAGE_USERS,INTERACT_ACROSS_USERS', 'ACCESS_COARSE_LOCATION,ACCESS_FINE_LOCATION']
0.7723	[isUserUnlocked, setExact]	['MANAGE_USERS,INTERACT_ACROSS_USERS', 'SCHEDULE_EXACT_ALARM']
0.7486	[setExact, getLastKnownLocation]	['SCHEDULE_EXACT_ALARM', 'ACCESS_COARSE_LOCATION,ACCESS_FINE_LOCATION']
0.7449	[requestAudioFocus, getNetworkType]	['MODIFY_PHONE_STATE', 'READ_PHONE_STATE']
0.7162	[requestAudioFocus, getLastKnownLocation]	['MODIFY_PHONE_STATE', 'ACCESS_COARSE_LOCATION,ACCESS_FINE_LOCATION']
0.7116	[isUserUnlocked, setExact, getLastKnownLocation]	['MANAGE_USERS,INTERACT_ACROSS_USERS', 'SCHEDULE_EXACT_ALARM', 'ACCESS_COARSE_LOCATION,ACCESS_FINE_LOCATION']
0.7097	[requestAudioFocus, isUserUnlocked]	['MODIFY_PHONE_STATE', 'MANAGE_USERS,INTERACT_ACROSS_USERS']
0.7029	[getNetworkType, getLastKnownLocation]	['READ_PHONE_STATE', 'ACCESS_COARSE_LOCATION,ACCESS_FINE_LOCATION']
0.6966	[requestAudioFocus, setExact]	['MODIFY_PHONE_STATE', 'SCHEDULE_EXACT_ALARM']
0.6950	[getNetworkType, isUserUnlocked]	['READ_PHONE_STATE', 'MANAGE_USERS,INTERACT_ACROSS_USERS']
0.6870	[setExact, getNetworkType]	['SCHEDULE_EXACT_ALARM', 'READ_PHONE_STATE']
0.6594	[requestAudioFocus, getNetworkType, getLastKnownLocation]	['MODIFY_PHONE_STATE', 'READ_PHONE_STATE', 'ACCESS_COARSE_LOCATION,ACCESS_FINE_LOCATION']
0.6560	[isUserUnlocked, setExact, requestAudioFocus]	['MANAGE_USERS,INTERACT_ACROSS_USERS', 'SCHEDULE_EXACT_ALARM', 'MODIFY_PHONE_STATE']
0.6558	[requestAudioFocus, getNetworkType, isUserUnlocked]	['MANAGE_USERS,INTERACT_ACROSS_USERS', 'READ_PHONE_STATE', 'MANAGE_USERS,INTERACT_ACROSS_USERS', 'MODIFY_PHONE_STATE']
0.6517	[requestAudioFocus, setExact, getNetworkType]	['MODIFY_PHONE_STATE', 'SCHEDULE_EXACT_ALARM', 'READ_PHONE_STATE']
0.6465	[requestAudioFocus, isUserUnlocked, getLastKnownLocation]	['MODIFY_PHONE_STATE', 'MANAGE_USERS,INTERACT_ACROSS_USERS', 'ACCESS_COARSE_LOCATION,ACCESS_FINE_LOCATION']
0.6464	[getNetworkType, setExact, isUserUnlocked]	['READ_PHONE_STATE', 'SCHEDULE_EXACT_ALARM', 'MANAGE_USERS,INTERACT_ACROSS_USERS']
0.6441	[hasEnrolledFingerprints, isUserUnlocked]	['USE_FINGERPRINT,INTERACT_ACROSS_USERS', 'MANAGE_USERS,INTERACT_ACROSS_USERS']
0.6368	[requestAudioFocus, setExact, getLastKnownLocation]	['MODIFY_PHONE_STATE', 'SCHEDULE_EXACT_ALARM', 'ACCESS_COARSE_LOCATION,ACCESS_FINE_LOCATION']
0.6327	[getNetworkType, isUserUnlocked, getLastKnownLocation]	['READ_PHONE_STATE', 'MANAGE_USERS,INTERACT_ACROSS_USERS', 'ACCESS_COARSE_LOCATION,ACCESS_FINE_LOCATION']

TABLE III
TOP 20 HIGH-SUPPORT ITEMSETS WITH MINIMAL SIZE OF 2

id	-
com.mnjgames.kashmiriweddinglove.indianarrangemarriage	com.irmoesbrodi.bolso
com.used.aoe	com.aokey
its.BoBG.MinecrashAdventure	sami.pro.rendomchat
com.zeeron.callbreak	com.mobi2fun.bangaloremetro
genesis.nebula	com.mitosisgames.match3
com.mcp.amngusmod	ro.ascendnet.bringo
com.flix.vpn	air.com.bigwigmedia.penguindiner2
com.magicgames.toyarmydrawdefense	bg.mobio.angeltarot
happypt.knk.AudacityLite.shortcuts	com.yasapets.island
com.kawaii.megumi	com.razzlepuzzles.sudoku

TABLE IV
APP IDS IN CASE STUDY

```
.line 62
:try_start_0
invoke-virtual {p0},
    Landroid/telephony/TelephonyManager;
    ->getDeviceId()Ljava/lang/String;

move-result-object p0
:try_end_0
```

Fig. 8. Example `conditional` code block

quest only necessary permissions in the manifest file, they also include ads, proxy, and analytic thirdy-party SDKs to monetize their app. These third-party SDKs are often opportunistic as they do not explicit ask for user's permissions, but instead they include *conditional* code blocks. These `conditional` code blocks are executed if and only if the app holds specific permissions and take advantages of every permission which their host apps are bing granted.

We showcase an example of such `conditional` code block founded in *air.com.bigwigmedia.penguindiner2/com/gameanalytics* and illustrated in Figure 8. The example code includes a try-catch block that attempts to read the device's global unique identifier (IMEI). If the host app has the necessary permission, `READ_PRIVILEGED_PHONE_STATE`, the third-party SDK would also be able to access the unique identifier. However,`READ_PRIVILEGED_PHONE_STATE` is reclassified as a system-level permission and the `getDeviceId` method is marked deprecated after API level 24. This also suggests that many inferred system-level permissions in our API instances are due legacy third-party SDK code that has not been updated.

Additionally, we counted 17 out of 20 apps are using at least one third-party SDKs to monetize their apps while 3 other apps adapted code obfuscation techniques and make it difficult to verfiy the use of third-party SDKs. The top three commonly used third-party SDKs are google(10), unity(7), and facebook(4).

B. AlarmManager and LocationManager

As Section V-D, we identified common usage patterns for different APIs and permissions. The FP-growth algorithm uncovered the intimacy of `AlarmManager` and `LocationManager`. To support this finding, our inspection found a concrete example which coupled of `AlarmManager` and `LocationManager` in `sami.pro.rendomchat.dpk`, and we present a simplified version in Figure 9. The reader can read a more detailed version in Figure 12 in the Appendix.

As shown in Figure 9, the app initializes an instance of AlarmManager and a location request Intent. By feeding the location request Intent to the instianated AlarmManager, the app gains the capability to request the location information at specified times. As aforementioned, holding `SET_EXACT_ALARM` and `ACCESS_FINE_LOCATION` at the same time allow an

```
alarmManager := AlarmManager()
locationManager: = LocationRequest()
alarmManager.setTask(locationManager)
```

Fig. 9. Simplified coupling of `AlarmManager` and `LocationManager`

app to periodically query the user's location. While solely holding `ACCESS_FINE_LOCATION` would allow the app to perform the same periodically requery by querying at regular intervals, the extra functionality from AlarmManager enables the app to query the location in more specific and targeted points in time. For example, querying the user's location at midnight would likely reveal the location of the user's home as the user is likely being sleeping at home at that time. Such approach allows apps to obtain more information with fewer queries, making it less likely to to be detected and keep the location querying low-profiler. It' worth nothing that granting new permissions is not simply an addition to set of operations can be performed. Rather, it is more like multiplication or exponentiation on the original set of operations, which enables unforeseeable and innovative attacks.

VII. Limitations and Future Work

On top of our existing observations and findings, we also see many potential areas and future work directions to expand our study.

A. Android Permission Annotations

Our current permission map is based on the `RequiresPermission` annotations. However, Android developers do not necessarily need to follow the convention as the annotations do not affect the semantics of the source code. As a result, our current permission map is incomplete. Moreover, Android development has not imposed strict checks to ensure that the current `RequiresPermission` annotations in the Android source correctly reflect the permissions. A method may be annotated with irrelevant permission but needs different permission to execute. Thus, our permission map is not sound.

Our permission mapping is dependent on accurate annotations from Android developers. While previous state-of-the-art permission mappings [16], [17], [22] have made progress in improving the soundness of the permission map, are outdated and only support API-level below 25. Furthermore, the most-recent state of art permission mapping [22] failed to released its code as promised, hindering the ability for researchers to build upon its foundation. Since conducting research for a new state-of-the-art mapping is not the focus of our study, we chose to use the permission annotations for our approach.

B. Frequent Itemset Mining

Our focus in analyzing API usage protected by permissions has primarily been on 100 itemsets with high support (above 0.5). However, it may be beneficial to consider lower support values and analyze more itemsets, as not all malicious behaviors are common among all apps. A itemset with lower support may reveal specific malicious behaviors or security concerns that would otherwise go unnoticed. For example, lowering the minimum support value to 0.2 increases the number of itemsets to 2,312, resuling a enlarged number of itemsets includes extremely sensitive APIs such as getDeviceID and getImei that are not captured in the current 100 itemsets. Moreover, a support value of 0.2, which is still considered reasonable, supports our objective of evaluating the Android permission system in its entirety.

C. Case Study & Static Analysis

During the user study section, we evidenced the existence of the coupling between `AlarmManager` and `LocationManager`, which we discussed in detail, including a potential attack that could exploit this vulnerability. However, our study suffers from the fact we used static analysis to observe how the sampled app was using these two components. We did not monitor the usage of coupling `AlarmManager` and `LocationManager` through dynamic analysis, which could provide more realistic understanding of the permission system. For example, one might build log system and monitor the usage of the code block in Figure 9.

Additionally, our case study requires extensive amount of manual efforts to analyze each line of the Java byte code, which could fail when apps adapt code obfuscation techniques.

To address the limitations, Future research could extend our approach and adapt dynamic analysis approach to better understand the android permission system in a more realistic way. For example, Reardon [5] performed instrumented dyanmic analysis and guided apps to reveal their malicious behaviors, uncovering multiple side channels and convert channels.

VIII. Conclusion

In conclusion, this paper provides insights into the Android permission system through a static analysis of its design and implementation. An up-to-date API-permission mapping was constructed to determine the permissions needed to access Android framework API. 35,117 apps were collected and analyzed, with the results showing that over 99% of the apps request internet access in their manifest files, even though many of them have no apparent reason for it. The analysis of these apps' usage of permission-guarded APIs through frequent itemset mining revealed a common pattern of the use of `AlarmManager` with `LocationManager` to periodically obtain the user's location. We discussed the potential implications for such coupling could make the location query stealthy and more targeted. We discussed the potential security implications of this coupling and validated our hypothesis with a detailed user study.

Our observations demonstrated that granting permissions to apps enables an exponential increase in the set of operations that the apps can perform, rather than just an addition of operations. Additionally, adding permissions enables many innovative attacks that were not easily foreseen. We identified the descrpiencies between the permissions requested in the manifest files and the inferred permissions from API call

instances. These inferred permissions were more privileged, which we attribute to the opportunistic data stealing behaviors of third-party SDKs. The findings from this study have the potential to guide future research and offer valuable understanding of the Android permission system.

Based on our findings, it can be concluded that the Android permission system is not well-designed and we urge future research to identify new ways to solve the existing design flaws that we stated above.

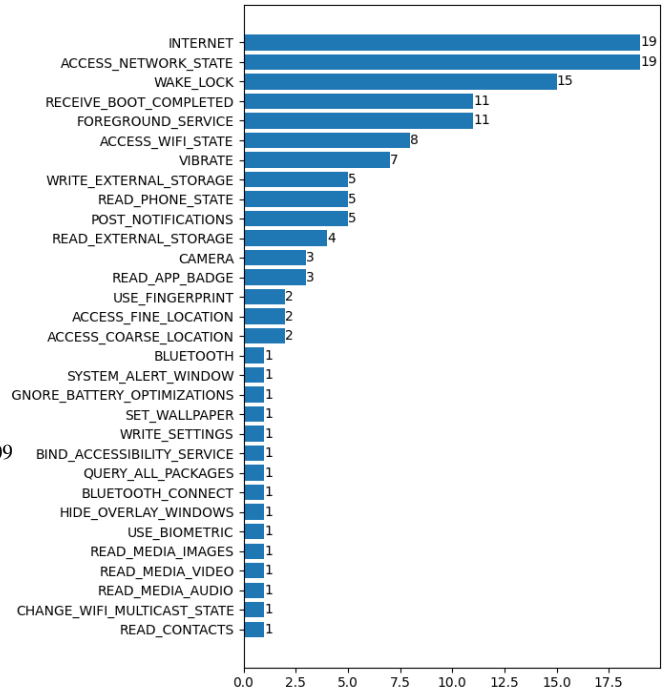

Fig. 10. Top 20 most used permissions to protect API calls

REFERENCES

[1] statcounter, "Android market share," https://gs.statcounter.com/os-market-share/mobile/worldwide, 2022, online; accessed Nov 30, 2022.

[2] statista, "Android market share," https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009 2009, online; accessed Nov 30, 2022.

[3] Z. Lei, Y. Nan, Y. Fratantonio, and A. Bianchi, "On the insecurity of sms one-time password messages against local attackers in modern mobile devices," in *Network and Distributed Systems Security (NDSS) Symposium 2021*, 2021.

[4] J. Wang, Y. Xiao, X. Wang, Y. Nan, L. Xing, X. Liao, J. Dong, N. Serrano, H. Lu, X. Wang *et al.*, "Understanding malicious cross-library data harvesting on android." in *USENIX Security Symposium*, 2021, pp. 4133–4150.

[5] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the android permissions system," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 603–620.

[6] J. Kim, J. Park, and S. Son, "The abuser inside apps: Finding the culprit committing mobile ad fraud." in *NDSS*, 2021.

[7] X. Mi, S. Tang, Z. Li, X. Liao, F. Qian, and X. Wang, "Your phone is my proxy: Detecting and understanding mobile proxy networks," in *Proceeding of ISOC Network and Distributed System Security Symposium (NDSS), 2021*, 2021.

[8] G. L. Scoccia, A. Peruma, V. Pujols, I. Malavolta, and D. E. Krutz, "Permission issues in open-source android apps: an exploratory study," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2019, pp. 238–249.

[9] S. Wu and J. Liu, "Overprivileged permission detection for android applications," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–6.

[10] Y. Wang, Y. Wang, S. Wang, Y. Liu, C. Xu, S.-C. Cheung, H. Yu, and Z.-l. Zhu, "Runtime permission issues in android apps: Taxonomy, practices, and ways forward," *IEEE Transactions on Software Engineering*, 2022.

[11] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1039–1055.

[12] J. Gamba *et al.*, "" do android dream of electric sheep?" on privacy in the android supply chain," Ph.D. dissertation, Universidad Carlos III de Madrid, Spain, 2022.

[13] R. Shawaga, "Android API Permission Lookup," https://evolving-android.cpsc.ucalgary.ca/index.html, 2019, online; accessed Sep 20, 2022.

[14] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 31–40.

[15] I. M. Almomani and A. Al Khayer, "A comprehensive analysis of the android permissions system," *IEEE Access*, vol. 8, pp. 216 671–216 688, 2020.

[16] 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*, 2011, pp. 627–638.

[17] 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*, 2012, pp. 217–228.

[18] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM sigmod record*, vol. 29, no. 2, pp. 1–12, 2000.

[19] G. Developers, "Documentation for app developers ," https://developer.android.com/docs, online; accessed Sep 23, 2022.

[20] L. Torvalds, "Linux kernel," https://github.com/torvalds/linux, 1991, online; accessed Dec 1, 2022.

[21] A. O. S. Project, "Android Open Source Project," https://source.android.com/, online; accessed Dec 1, 2022.

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

[23] M. Al Ali, D. Svetinovic, Z. Aung, and S. Lukman, "Malware detection in android mobile platform using machine learning algorithms," in *2017 International Conference on Infocom Technologies and Unmanned Systems (Trends and Future Directions)(ICTUS)*. IEEE, 2017, pp. 763–768.

[24] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé *et al.*, "Deep android malware detection," in *Proceedings of the seventh ACM on conference on data and application security and privacy*, 2017, pp. 301–308.

[25] ibotpeaches, "Apktool," https://ibotpeaches.github.io/Apktool/contribute/, online; accessed Nov 30, 2022.

[26] GitHub, "dex2jar," https://github.com/pxb1988/dex2jar, 2011, online; accessed Sep 21, 2022.

[27] E. Alepis and C. Patsakis, "Unravelling security issues of runtime permissions in android," *Journal of Hardware and Systems Security*, vol. 3, no. 1, pp. 45–63, 2019.

[28] Y. Shen, P.-A. Vervier, and G. Stringhini, "Understanding worldwide private information collection on android," *arXiv preprint arXiv:2102.12869*, 2021.

[29] Y. Chen, X. Jin, J. Sun, R. Zhang, and Y. Zhang, "Powerful: Mobile app fingerprinting via power analysis," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[30] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 447–458.

APPENDIX

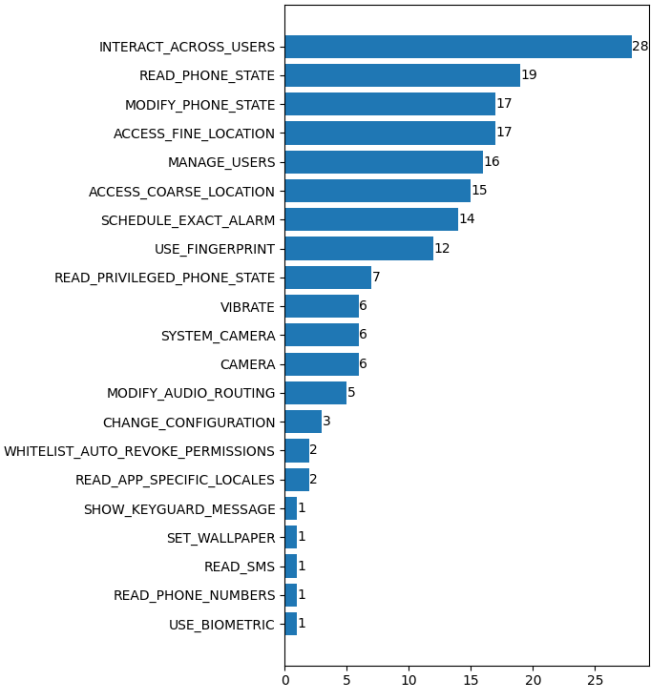

Fig. 11. Top 20 most requested permissions in collected manifests

```
.method synthetic constructor <init>
(Lo/AlarmManager$AlarmClockInfo;
Lo/RemoteInput;
Lo/Location;
Lo/ProviderInfo;
Lo/WallpaperColors;)V:

...
invoke-direct {p0},
 Ljava/lang/Object;-><init>()V

...

new-instance v1,
 Lo/LocationRequest;

invoke-direct {v1, p3},
 Lo/LocationRequest;-><init>(Lo/Location;)V

iput-object v1, p0,
 Lo/getForegroundServiceType;->write:Lo/zi;
```

Fig. 12. Detailed example of AlarmManager and LocationManager coupling