Small Changes, Big Changes: An Updated View on the Android Permission System*

Yury Zhauniarovich[1] and Olga Gadyatskaya[2]

[1] Qatar Computing Research Institute, HBKU, Qatar
[2] SnT, University of Luxembourg, Luxembourg

Abstract. Since the appearance of Android, its permission system was central to many studies of Android security. For a long time, the description of the architecture provided by Enck et al. in [31] was immutably used in various research papers. The introduction of highly anticipated runtime permissions in Android 6.0 forced us to reconsider this model. To our surprise, the permission system evolved with almost every release. After analysis of 16 Android versions, we can confirm that the modifications, especially introduced in Android 6.0, considerably impact the aptness of old conclusions and tools for newer releases. For instance, since Android 6.0 some signature permissions, previously granted only to apps signed with a platform certificate, can be granted to third-party apps even if they are signed with a non-platform certificate; many permissions considered before as threatening are now granted by default.
In this paper, we review in detail the updated system, introduced changes, and their security implications. We highlight some bizarre behaviors, which may be of interest for developers and security researchers. We also found a number of bugs during our analysis, and provided patches to AOSP where possible.

Keywords: Android security, permission system, runtime permissions, compatibility challenges

1 Introduction

Nowadays, Android is the dominating smartphone operating system. It occupied more than 80% of the total smartphone market share in 2015 [20]. Furthermore, Android is truly ubiquitous existing in the Auto, TV, and Wear flavors. Moreover, many other types of devices, e.g., cameras or game consoles, run tweaked Android firmware [17]. Overall, more than 1.4 billion active devices are currently powered by Android [14]. This huge user-base was achieved by Google thanks

* We thank the anonymous reviewers for their comments that allowed to improve the paper. We are also very grateful to William Enck for shepherding the paper and suggesting many improvements to it. The work of Olga Gadyatskaya was supported by the Luxembourg National Research Fund (C15/IS/10404933/COMMA).

to, among all, frequent updates of the operating system that keep introducing new features and improving performance.

Yet, the wide landscape of device types and platform versions gives rise to compatibility challenges. While the latest devices are relatively well-updated, others can be left behind, or even never updated after the release. For instance, Google reported that 2.6% of devices that had visited the Google Play Store in March 2016 ran Android 2.3 released in 2011 [13]. At the same time, third-party applications are typically updated frequently, yet some of them are unsupported by the developers after a while. Therefore, there is a high fragmentation of the eco-system, and many problems, including security ones, emerge due to discrepancies in update cycles of the platform and apps.

The Android permission system regulating access of apps to device capabilities and system components, such as telephony, file system, sensors, networks, etc., is a crucial part of the Android security model. Not surprisingly, from the beginning it was, and still is, central to many studies of Android security (it is featured in [23, 25, 28, 31, 33, 35, 41, 45, 46, 49, 50, 52, 53], to mention a few). However, only some of them acknowledged that the permission system was not stable. Among those, an early investigation by Enck, Ongtang and McDaniel [31] reported on the substantial shift introduced to the permission system across the earliest Android releases. Since that time, the vast majority of Android studies still rely on the same understanding introduced in this seminal paper.

The **emergence of runtime permissions in Android 6.0** forced us to take a closer look at the permission system design. In this paper we analyze the changes in the permission system introduced in the last 6 years and provide an updated view on the current architecture of the Android permission system since its description in [31]. We reveal the core changes that need to be considered during the security analysis, the main of which are the following:

- **Runtime permissions.** In Android 6.0, permissions are divided into install-time and runtime. *Normal* and *signature* (with some exceptions) permissions are permanently assigned upon the app installation, while *dangerous* permissions are now granted at runtime, and the user may revoke them at any time.
- **Runtime permissions are granted on the group basis.** If an app requires runtime permissions related to the same permission group, once one of them is granted, others are granted as well. Instead of enabling more fine-grained control of dangerous functionality, Android 6.0 does the opposite.
- **Some *signature* permissions can be obtained by third-party applications.** The Android community is used to consider *signature* permissions to be install-time granted to apps that have the same digital signature as the package declaring the permission. However, several new types of *signature* permissions appeared in Android that can be obtained by third-party apps not conforming to this condition.
- **The *signature|system* protection level is deprecated.** Currently, the *signature|system* protection level is marked as deprecated and should not be used neither for custom (third-party), nor for platform permissions.

– **Some dangerous permissions are now granted without user's consent.** In Android 6.0, 22 permissions, previously considered as sensitive, are granted by default and the user cannot revoke them in any way. For instance, the `INTERNET`, `BLUETOOTH`, `NFC` permissions are now automatically granted at app installation. Previously they had to be approved by the user.

Considering the aforementioned modifications, it is clear that the Android community needs to update its view on the permission system and to evaluate security implications of the changes. To address this need, in this paper we present an updated security architecture of the system and important internal details of its implementation. Furthermore, to assess the compatibility challenges implications, we performed a thorough longitudinal study of the Android permission system that yielded many **interesting findings**, e.g.,:

– Even though the *signature|system* protection level is deprecated, permissions of this level still exist in the system. Moreover, 9 permissions of this type were added in the Android 6.0 release itself. We have submitted to Google several patches to fix this issue in Android Open Source Project (AOSP), and some of them have already been merged into the master branch.
– The runtime permissions have backward compatibility issues. Developers that expect their apps to run on older platform versions are still required to make a runtime check for permissions. However, the permissions that did not exist on some platform version are always denied (while they should not be required at all). We have found 8 such permissions, e.g., `ADD_VOICEMAIL`.
– Some non-*dangerous* permissions are assigned to permission groups, although there is no reason for this. We found 8 such permissions, e.g., `USE_FINGERPRINT`. We consider these to be coding nits that could be fixed by Google developers.

Our findings emphasize considerable flaws that emerged due to the high change rates in the permission system design. Considering the aforementioned discrepancies in update cycles of platforms and apps, it is time for the security community to re-evaluate the attack surface of the Android permission system.

Roadmap. §2 outlines the established view on the Android permission system. §3 incrementally updates this view, while §4 gives internal details of the permission system implementation. §5 presents our quantitative analysis of evolution in the permission system, and §6 presents the key findings of our qualitative study. Finally, §7 discusses related work, and §8 concludes the paper.

2 The Established View on the Permission System

By default, all Android apps are executed as low-privileged processes at the Linux kernel level. Thus, every app has access only to a limited set of system capabilities. At the same time, to be fully-functional an app should be able to interact with other applications and obtain data from various system services (e.g., location or telephony) running in other processes. To enable these interactions, Android provides a special inter-component communication (ICC)

protocol called Binder. Certainly, these communications should not be arbitrary, i.e., only approved interactions must be possible within the system. The Android permission system provides such access control mechanism. Permissions, which are unique security labels, are assigned to sensitive resources. Once an app is granted with the permission, it receives access to the corresponding protected object, otherwise interactions with the resource are prohibited.

A permission must be declared by the developer in the `AndroidManifest.xml` file of the app (in the special `permission` tag) and assigned to the protected resource (either in the manifest file or by performing corresponding checks in the code). Once declared, other packages may ask for access to the object by requesting the corresponding permission using the `uses-permission` tag of their own `AndroidManifest.xml` file. *Platform* permissions are declared within the Android operating system itself: either in the Android framework or in the packages supplied with the platform. Third-party app developers may also declare their own *custom* permissions and use them to protect sensitive components of their apps.

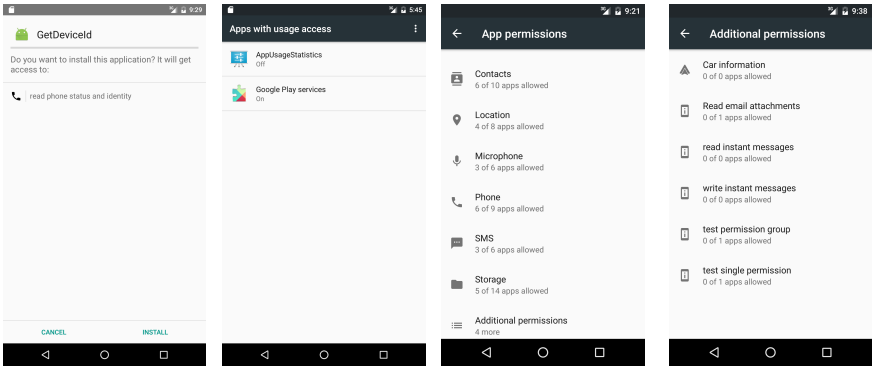
Upon declaration, any Android permission is assigned with a *protection level*. It defines what apps can be granted with the corresponding permissions, and how this process occurs. Starting with Android 0.9 [31], permissions were divided into 4 levels: *normal*, *dangerous*, *signature* and *signature|system*. According to the established permission system view, the least sensitive *normal* permissions were granted automatically to any app declaring these permissions, while more sensitive *dangerous* permissions were granted only after user's explicit consent during app installation. If the user wanted to refuse even a single permission, the application would not be installed on the device. *Signature* permissions were granted only if packages declaring the permission and using it are signed with the same certificate. Finally, permissions of the *signature|system* protection level acted like *signature* permissions, but could be additionally granted to apps installed into the system partition. Thus, prior to Android 6.0 all permissions were granted or denied once and for all at the installation time.

A permission can belong to a *permission group* that clusters together security labels according to particular functionality. Permission groups were mainly introduced to simplify the presentation by grouping permissions together. Yet, before Android 6.0 groups were not widely adopted in the "vanilla" Android, although they were used in the Google Play client application.

This vision of the Android permission system migrated for a long time from one research paper to another. In the meanwhile, the system did not stand still, but continuously changed all that time. However, the modifications were not that crucial, and remained mostly unnoticed.

3 New Android Permission System Overview

In Android 6.0, **all permissions are divided into installation and runtime**. Roughly, this division occurs in the following way: *normal*, *signature* and *signature|system* permissions are permanently granted upon the app installation (yet,



(a) (b) (c) (d)

Fig. 1: Screenshots: a) Permission request during installation of legacy applications in Android 6.0; b) Screen to grant or revoke "appop" permission; c) Separate screens are developed for core permission groups to grant and revoke permissions; d) List of additional permissions.

with some exceptions considered further), while *dangerous* permissions are now checked at runtime. The *signature|system* protection level is **deprecated** starting Android 6.0 and should not be used [12]. However, our analysis of permissions defined in the platform code shows that such permissions are still abundant (see Sec. 5 and Sec. 6 for more details).

Previously, *dangerous* permissions were to be approved by the user in the special screen shown during app installation. Once approved, the app could be instantly used and the user did not deal with permissions anymore. In Android 6.0, the screen to grant *runtime* permissions is not shown (for apps targeting API 23 and up). Instead, all *runtime* permissions after installation are in the *disabled* state and must be approved by the user once the app needs access to the protected functionality.

To support runtime permissions, special protected API calls were added to `PackageManager` allowing to grant and revoke permissions dynamically. Additionally, new APIs were added allowing app developers to check at runtime if permissions are granted and to request them if necessary [19]. Within the *Settings* app, the users are provided with two screens to review, grant and revoke runtime permissions: on the first screen permissions are grouped on per app basis, on the second – per permission group.

Obviously, new applications must be forward compatible with the older Android versions, because only a small fraction of devices runs the newest Android (in April 2016 only 5% of devices ran Android 6.x [13]). To ensure compatibility, Google provided a special compatibility library that proxies the calls for checking granted permissions (`ContextCompat.checkSelfPermission`). However, this proxy call must still rely underneath on the permission check functionality available in the previous releases, which, not surprisingly, is based on

the `Context.checkPermission` API call. In previous Android versions permissions are granted upon installation, thus, the check will always pass, and new runtime permission request functionality will not be called. However, we found out that this functionality does not always work as expected (see Sec. 6).

Backward compatibility of legacy apps with the new version of Android is provided through the *AppOps* system allowing users to grant and revoke permissions at runtime through a dedicated user interface within the *Settings* system application. It shares the same interface with the runtime permission manager. This hidden app permission manager unofficially appeared in Android 4.3. Unfortunately, access to this component was suppressed in Android 4.4.2 and reappeared only in Android 6.0. However, *AppOps* handles only platform permissions and, thus, cannot enforce custom *dangerous* permissions declared by a developer. Upon installation of a legacy app through the installer on device the user is still presented with the "old" grant permission screen (see Fig. 1a). The user must agree with the presented permissions, or the app will not be installed. This behavior differs from the one of the apps targeting Android 6.0, what results in some user experience inconsistencies. We describe them in details in Sec. 6.

Runtime permissions are granted per **permission groups**, i.e., if one permission from a group is granted or revoked, the same happens for all permissions in this group. For instance, if an app is granted with the `READ_CONTACTS` permission, it automatically receives `WRITE_CONTACTS` (if requested), because they both belong to the `CONTACTS` permission group. Android 6.0 defines nine permission groups for *dangerous* permissions: `CALENDAR`, `CAMERA`, `CONTACTS`, `LOCATION`, `MICROPHONE`, `PHONE`, `SENSORS`, `SMS`, `STORAGE`. While the app developers still have to declare permissions from these groups individually, the end-users only grant or revoke access per permission groups, and they are oblivious to which individual permissions the app requests.

Before it was assumed that third-party applications cannot obtain any *signature* permission if they are not signed with the same certificate. Yet, in Android 6.0 new permissions called *appop* were added. These *signature* permissions (`PACKAGE_USAGE_STATS`, `WRITE_SETTINGS` and `SYSTEM_ALERT_WINDOW`) can now be granted to third-party apps after an explicit user's consent through *Settings*.

We continue to explore the changes to the Android permission system and their implications for security analysis in Sec. 6.

4 Permission System Implementation Details

The behavior of permissions is controlled through assigning special string values to the attributes (`android:protectionLevel` and `android:permissionFlags`) upon permission declaration in the `AndroidManifest.xml` file. During the installation of a package, these values are parsed influencing on the bits of two 32-bit integer fields (`protectionLevel` and `flags`) of the `PermissionInfo` class. This section reviews how the bits of these two fields affect the permissions behavior.

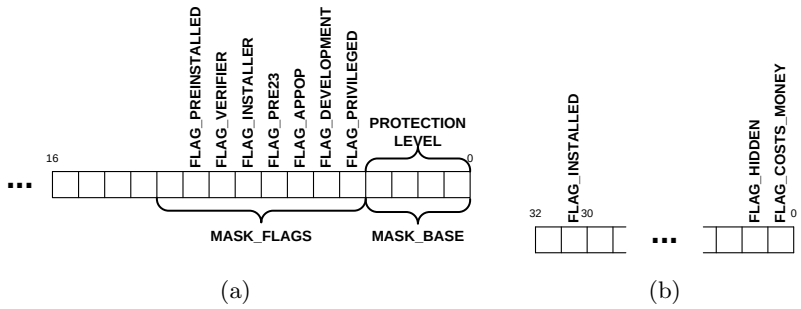


Fig. 2: "Protection Level" (2a) and "Additional Flags" (2b) field map.

4.1 Protection Level

Fig. 2a shows a map of the lower 16 bits of the `protectionLevel` field (the higher 16 bits are currently not in use). The lower 4 bits are used to specify the protection level of a permission. The protection level value is determined by applying bitwise `AND` operation to the `protectionLevel` field and the `MASK_BASE` constant. Since a permission can only have one protection level, its values have sequential order, where the *normal* protection level is equal to 0, *dangerous* is 1, *signature* – 2, and *signature|system* is equal to 3. Interestingly, although *signature|system* level has a higher protection level value, *signature* permissions are considered as more sensitive. If a permission protection level is not specified in the manifest file, by default, *signature* protection level is used.

Protection level flags can be used only with *signature* permissions. Flags with other protection levels will generate an error at the time of manifest parsing. Protection level flags are masked with the `MASK_FLAGS` constant.

The first flag `FLAG_PRIVILEGED` enforces that only apps either signed with the same certificate or installed into the special location can obtain the permission. Until Android 4.4 all applications installed on the system image were granted with these privileged permissions by default. This means that even unprivileged system apps, e.g., Calculator, were able to obtain such permissions. To reduce the attack surface, system apps were later divided into the *ordinary* and *privileged* ones [9]. The ordinary system apps remain in the `/system/app` directory, but **are not** granted with privileged permissions anymore. To obtain privileged permissions an app must be installed into a separate `/system/priv-app` folder. Besides setting this flag directly, the developer can achieve the same permission behavior by setting the protection level to *signature|system* (deprecated since Android 6.0).

In Android 4.1 [12], the *development* permissions (marked with the flag `FLAG_DEVELOPMENT`) were introduced. These permissions usually protect the functionality required to perform development tasks, e.g., read system logs (`READ_LOGS`). An app can request these permissions, but they will not be automatically granted. At runtime the user can grant and revoke these permissions on demand by using special commands `pm grant` and `pm revoke` of the Android shell [30].

FLAG_APPOP was introduced in Android 5.0 [4], although explicitly it has started to be used only with Android 6.0. This flag was added to allow selective access to certain critical platform operations protected by *signature* permissions to third-party apps, after an explicit approval from the user. As we mentioned, typically, the *signature* protection level ensures that the corresponding platform permission is automatically granted at install time to the apps signed with the same certificate as the system image. Yet, this flag relaxes the requirement that the protected functionality can be used only by the system components. However, upon installation, access to the resources is disabled by default to third-party apps. For every permission of this *appop* type there is a separate configuration screen, where the user may explicitly grant or revoke access to these operations for system and third-party apps. E.g., Fig. 1b shows the screen for the PACKAGE_USAGE_STATS permission. The flag name shows that the enforcement of this type of permissions happens through the *AppOps* system.

FLAG_PRE23, as the name suggests, indicates that the corresponding permission is automatically granted to apps targeting pre-6.0 Android (API levels less than 23) versions [11]. For instance, the permission to draw a window over other apps SYSTEM_ALERT_WINDOW before Android 6.0 had the *dangerous* level, and thus was granted automatically upon installation. In Android 6.0 the protection level of this permission was changed to *signature*. However, apps targeting previous API versions are not aware of this change. Thus, during their execution an invocation of the functionality protected with this permission will generate an error. FLAG_PRE23 permits to overcome this issue by automatically granting the permission with this flag set to apps targeting previous versions of Android.

The flags FLAG_INSTALLER and FLAG_VERIFIER introduced in Android 6.0 [5] indicate that permissions are automatically granted to the packages set as the required installer and verifier (see more in [30]). However, to use these permissions the installer package must be installed on the system image, while the verifier package must be additionally granted with the PACKAGE_VERIFICATION_AGENT permission which has the *signature*|*privileged* protection level.

Finally, FLAG_PREINSTALLED added in Android 6.0 [8] indicates that the permission can be granted not only to the apps installed into the privileged folder, but to any app installed in the system partition.

4.2 Permission Flags

Permission flags were introduced in Android 4.2 [3]. Internally, permission flags are also represented as an integer 32-bit field which map is shown in Fig. 2b. These flags are controlled through the android:permissionFlags attribute of the permission tag. It should be noted that only the FLAG_COSTS_MONEY and FLAG_HIDDEN flags may be set through this attribute, while FLAG_INSTALLED is not accessible through a permission declaration.

The flag FLAG_COSTS_MONEY introduced in Android 4.1 [3] influences how a permission with this flag set is presented to a user. These permissions are marked with the "coins" sign displayed on the screen shown during app installation (in

Table 1: Versions of the Android platform used in our study

API level	Branch	Codename	Release date (mm-dd-yyyy)
23	android-6.0.0_r1	Marshmallow	10-05-2015
22	android-5.1.0_r1	Lollipop	03-09-2015
21	android-5.0.1_r1	Lollipop	12-02-2014
19	android-4.4_r1	KitKat	10-31-2013
18	android-4.3_r1	Jelly Bean	07-24-2013
17	android-4.2_r1	Jelly Bean	11-13-2012
16	android-4.1.1_r1	Jelly Bean	07-11-2012
15	android-4.0.3_r1	Ice Cream Sandwich	12-16-2011
14	android-4.0.1_r1	Ice Cream Sandwich	10-21-2011
10	android-2.3.3_r1	Gingerbread	02-09-2011
9	android-2.3_r1	Gingerbread	12-06-2010
8	android-2.2_r1	Froyo	05-20-2010
7	android-2.1_r1	Eclair	01-12-2010
6	android-2.0.1_r1	Eclair	12-03-2009
5	android-2.0_r1	Eclair	10-26-2009
4	android-1.6_r1	Donut	09-15-2009

versions before Android 6.0). Interestingly, there are no restrictions on the usage of this flag, thus, even custom permissions could use it. Similarly, the flag `FLAG_HIDDEN` added in Android 6.0 [7] also influences presentation, making a permission hidden from the user's sight. This flag is used for the platform permissions that have become deprecated and removed from the system. However, a developer may use this flag to conceal custom dangerous permissions.

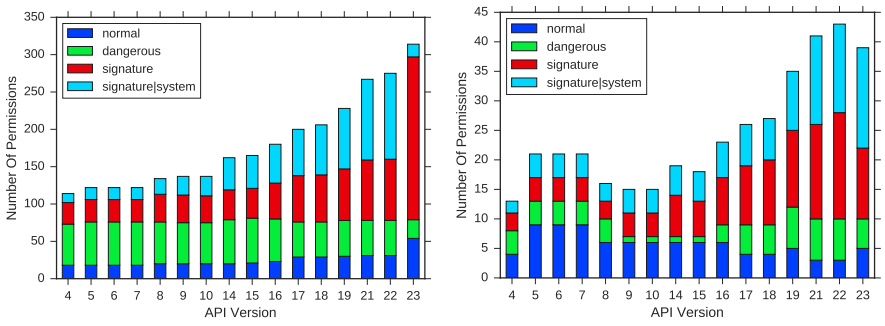
The flag `FLAG_INSTALLED` was introduced in Android 6.0 [10]. It is set by the operating system itself. This flag shows that the permission has been actually installed into the system, and influences presentation of permissions. For instance, if a permission has not been declared by an application but is requested by another app, it will not be shown in the list of requested permissions.

5 Analysis of Permission Changes

To investigate empirically how the Android permission system evolved across platform updates, we retrieved the source code of the Android platform for versions released from 2009 till 2015 that **resulted in the API level change** (the latest release at the time of writing is Android Marshmallow). Table 1 overviews the Android platform releases covered in our study.

We performed the analysis aiming at detection of odds in the permission system. In our study we focus on the Android platform permissions, and we do not cover custom permissions, which are defined by third-party applications to protect access to their resources. We divide platform permissions into 4 categories:

- *sample* – permissions that are declared by the sample apps shipped with the platform source code (appeared from API 21).
- *test* – permissions that are declared in the manifest files of packages developed for testing purposes;
- *package* – permissions that are declared in various packages that complement the framework, and that are not of *test* or *sample* groups;



(a) (b)

Fig. 3: Number of permissions for every platform release: a) for *core* permissions; b) for *package* permissions.

– *core* – permissions that are declared in the core Android manifest file located in the `frameworks/base/core/res` folder;

The categories discussed above reflect the basic purposes why permissions are used within AOSP [1]: some permissions (from the categories *core* and *package*) are the "true" permissions used for access control, while others are auxiliary utilized in example applications (*sample*) or for testing (*test*). We focus our study on *core* and *package* permissions, because they are the ones that truly influence the behavior of the operating system.

The study done by Wei et al. in 2012 revealed that the number of permissions steadily increased with each Android release [49]. Our study, as of the beginning of 2016, confirms that finding and shows that **the total amount of permissions declared within the Android platform continues to grow**, reaching 314 in API 23 compared to 165 in API 15 (the last version analyzed by Wei et al. [49]). Fig. 3a and Fig. 3b illustrate the growth of the number of permissions for *core* and *package* categories correspondingly. Obviously, the main contributor to the continuous increase are *core* permissions. The amount of the *package* permissions fluctuates, although still showing the overall upward trend. These plots also demonstrate the changes in the amounts of permissions of different protection levels. Table 2 characterizes the changes between consequent API levels. The data confirms that almost every Android API release (besides the API 6, 7, 10) introduced new permissions, as access to the new platform functionality often needs to be guarded.

Interestingly, while the total amount of permissions increases with every new Android release, the number of permissions with *normal* and *dangerous* levels, which guard the functionality exposed to third-party applications, remains fairly stable. Therefore, from the developer perspective, the cognitive load did not increase much in terms of new permissions (however, the amount of compatibility issues to be handled is still growing due to the fluctuations in permissions). At

Table 2: Permission changes in *core* and *package* categories

API level	Amount of permission changes					
	Added	Removed	Type changed	Protection level changed		
				inc	dec	total
5	14	2	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	13	2	1	5	1	6
9	8	6	0	4	0	4
10	0	0	0	1	0	1
14	30	1	0	5	1	6
15	3	1	0	0	0	0
16	20	0	0	6	0	6
17	21	0	0	7	8	15
18	10	1	0	0	0	0
19	28	2	0	2	0	2
21	54	9	0	6	3	9
22	11	3	0	0	0	0
23	46	8	0	7	128	135

the same time, security researchers, and platform and system app developers have to cope with more and more permissions.

At the same time, permissions are not only added. Throughout the platform evolution, many permissions were removed or changed their protection level. We analyzed code commits to AOSP [1] and found the following reasons why permissions are removed. Most of the *package* permissions were removed, because either the corresponding packages were deleted from the system, or the functionality of these packages became closed-source. Some permissions became obsolete because the corresponding functionality was either provided to all applications (e.g., the backup functionality protected with the `BACKUP_DATA` permission was made available to all apps in API 8) or merged with other functionality, as in case of `GRANT_REVOKE_PERMISSIONS` (removed in API 23) used to protect the runtime granting of *development* permissions. Interestingly, while the permission `READ_OWNER_DATA` was removed in API 9, more than 5 years ago, the current documentation still contains references to it[3]. Additionally, permissions may be simply renamed (e.g., `BROADCAST_SCORE_NETWORKS` became `BROADCAST_NETWORK_PRIVILEGED`). All these perturbations hinder understanding of the permission system and its changes across Android releases.

According to Table 2, there was only 1 case of the category change: the `ACCESS_CACHE_FILESYSTEM` permission in API 7 was in the *package* category, while in API 8 its declaration was moved to the core Android Manifest file.

As for the protection level changes, Table 2 reports the number of permissions that increased or decreased[4] their protection level.

The overall trend in the table shows that, prior to Android 6.0, permissions had a tendency to increase their protection level with the lapse of time. However,

[3] `http://developer.android.com/guide/topics/manifest/manifest-intro.html`

[4] For this table we interpret the protection levels *normal*, *dangerous*, *signature* and *signature|system* as an ordered set, where *normal* corresponds to the least critical permissions and *signature|system* – to the most critical.

the majority of protection level updates were related to changing the protection level from *signature* to *signature|system*, what is actually not a restriction in control. Although internally *signature|system* permissions are assigned with a higher value, in general the *signature* permissions are more restrictive, because they allow the apps to obtain these permissions only if the declaring and requesting packages are signed with the same certificate. Permissions of the *signature|system* level can be also granted if the app is installed into the special system folder, what allows vendors to use this functionality to vest pre-installed applications with additional capabilities. For instance, the ability to shutdown the system (protected with the `SHUTDOWN` permission) in API 14 was also given to vendor apps. At the same time, other changes of protection level mostly aimed at limiting the privileges of third-party apps. E.g., in API 16 the `READ_LOGS` permission allowing to read the system log that may contain sensitive data, changed level from *dangerous* to *signature|system*.

Before API 23 there were not so many cases of decreases in the protection level. These were mostly related to relaxing *dangerous* permissions in order to avoid bothering the end-users with their approval. For instance, the `WAKE_LOCK` permission allowing an app to prevent the system from going into the sleep mode changed its level from *dangerous* to *normal* in API 17.

There are permissions that changed their protection level several times. E.g., permission `BATTERY_STATS` initially appeared as *normal*. In API 17 it became *dangerous*, and in API 19 it emerged as a *signature|system* permission. Finally, in API 23 it became a *signature* permission. Thus, during its life `BATTERY_STATS` has had all possible security levels.

The API level 23 introduced significant changes in protection levels of permissions. Now, there are only a few *dangerous* permissions, as opposed to all previous Android releases. Table 2 shows that the protection level decreased for 128 permissions. The main reason for this change is deprecation of the *signature|system* protection level (104 permissions became *signature*). Moreover, the shift to runtime permissions forced platform developers to reconsider the entries in the *dangerous* set, leaving only the most critical ones that can be comprehended by users. Consequently, some *dangerous* permissions were transformed into *normal* (22 cases). Sec. 6 discusses the effects of these changes.

Permission groups show more stable behavior with respect to changes. In Android 1.6 (API 4) there were 11 groups. As permission groups were not widely used, this number remained the same till API 17, when 19 new groups were added. In API 18 one additional group appeared, resulting in 31 total. There is not much information why this reorganization happened in these 2 consecutive releases. However, this may be connected with the Google Play installer app starting to cluster permissions according to their groups [18]. In Android 6.0 permission groups were completely reconsidered once again. There are 4 new groups added, while 26 were removed, resulting in 9 groups total. This radical change happened because *dangerous* permissions are now granted on per-group basis. Thus, the amount of groups was considerably reduced to avoid overwhelming users with lots of permissions.

6 Key Findings

Ideally, the security critical components of a system should remain quite stable to ensure easy security assessment. Unfortunately, this does not hold true in case of the Android operating system. This section reports on our findings and doubts inferred during the analysis of the evolution of the permission system.

6.1 Important changes in API 23

1) Runtime permissions. Undoubtedly, from the security perspective one of the biggest changes in Android 6.0 is the introduction of runtime permissions. Such a change requires efforts from both the OS designers and third-party developers to ensure backward compatibility of old apps with the new platform version, and forward compatibility of new apps with older platforms.

Backward compatibility of old apps with the new platform. Although the intention was to make legacy (targeting the Android API levels before 23) and new (API level 23) apps to behave in the same way, the differences are quite substantial. First, during the installation of a legacy app the user must agree with the requested permissions, or it will not be installed (see Fig. 1a), while apps targeting API 23 will be installed silently. Second, after the installation all *dangerous* permissions of legacy apps will be in the granted state, while *runtime* permissions of new apps will be disabled. Third, and most important, in Android 6.0 only *core* permissions can granted and revoked to legacy apps, while if an app targets API 23 it is also possible to adjust custom *dangerous* permissions. Furthermore, some subtle differences require high attention from developers. For instance, developers must ensure that the application, which functionality is called, has been already granted with the permission to access this functionality [43]. Additionally, in order to use an external library, which requires access to the protected functionality, the developers must handle properly runtime permission requests [42].

Forward compatibility of new apps with older platforms. The new runtime permission functionality has not come transparently for the application developers. According to the new guidelines [19], before making an API call protected with a permission, the app should ascertain that the appropriate permission has been granted. If not, the developer must ask for the permission, and the user can allow or deny it. Irrespectively of the user's decision, both cases must be handled by the developer (see Sec. 3). Unfortunately, the check whether the permission has been granted does not always return the correct result. We found out that if a developer runs an app on the older Android version, which has not yet declared the requested permission, the permission check returns that the permission is denied, while actually it is not required. We made a script that automatically identifies the permissions producing this unexpected behavior by extracting the list of *runtime* permissions in Android 6.0, and comparing it with the lists of *dangerous* permissions in the previous versions. We found 8 such permissions added after API 4, namely `USE_SIP` (added in API 9); `ADD_VOICEMAIL` (in API 14); `WRITE_CALL_LOG`, `READ_CALL_LOG`,

READ_CELL_BROADCASTS, READ_EXTERNAL_STORAGE (in API 16); BODY_SENSORS (in API 21[5]); and READ_TV_LISTINGS (in API 23). These peculiarities are not described in the Android documentation, although some developers have started to experience problems[6]. At the same time, there is no bullet-proof solution how to overcome this issue at the operating system level (it is possible to implement the corresponding check in apps themselves [44]). As previous versions of Android are usually not supported (patches for older versions are rarely produced and deployed), it is practically impossible to deploy patches on all devices running older versions of Android. Handling through patching the Android support library is not a solution also, because developers may simply not use it in their apps. Thus, the developers must consider these cases in their applications themselves. In any case, this issue must be at least specified in the documentation.

2) Runtime permissions are granted per permission groups. Clearly, this decision was made to reduce the amount of interruptions for asking permissions at runtime and to facilitate user's understanding of permissions [36]. At the same time, experienced users are not given any option to control permissions in a more fine-grained manner. Similar functionality introduced for the first time in the Google Play client received negative feedback both from the users and security analysts [18]. Moreover, this architectural decision implies that security researchers have to consider permission groups in their analysis of apps.

We can remark here that for a long time security researchers have asked for better and more fine-grained control over sensitive data and functionality on Android (e.g., [32, 41, 45], to mention just a few). Android 6.0 clearly moves in the opposite direction. Arguably, the users often did not understand the implications of various *dangerous* permissions, and the reduced complexity of permissions could be beneficial for some end-users [36]. Therefore, new evaluations and studies of the system are required from the community.

3) UID sharing. There was an attempt to change permission granting to on per package basis. It failed, and permissions are still granted per UID [2]. This creates an additional attack possibility for collaborative applications sharing the same UID to access the functionality protected with *runtime* permissions. As we explained in Sec. 3, in Android 6.0 the screen with the required *runtime* permissions is not shown to the user during app installation, but the user's approval for these permissions is requested at the runtime. Thus, the user finds out about the required permissions only once they are requested. If two applications share the same UID, then if a user grants a *runtime* permission to one app, the second will be automatically granted with the same permission, and the user will be unaware of this fact. For instance, the Microsoft Excel [15] and Microsoft PowerPoint [16] apps share the same UID. Thus, if at runtime Microsoft Excel is granted with READ_EXTERNAL_STORAGE permission, the Microsoft PowerPoint app instantly receives the same permission even without user's consent. Addi-

[5] This permission was added in API 20, which we did not analyze (API 20 was developed for wearable systems).

[6] http://stackoverflow.com/questions/33482474/android-marshmallow-permission-model-on-os-4-0-read-external-storage-permission

tionally, the apps will also receive rights to perform the actions protected with the `WRITE_EXTERNAL_STORAGE` permission (if it is requested by the apps), because both permissions belong to the same group. This is clearly not the behaviour the user expects. The effort from the OS developers should be put into this direction.

4) Signature permissions available to third-party apps. Before it was assumed that third-party applications cannot obtain any *signature* permission if they are not signed with the same certificate. However, this is not true anymore, and any new security system for Android needs to take these permissions into account. In our analysis we found 4 groups of exceptions that considerably influence the security analysts. This change especially affects permission maps, which considered before only *dangerous* and *normal* permissions as available for third-party apps [23].

Appop permissions. Introduction of the *appop* permissions (with `FLAG_APPOP` set) entails quite substantial consequences. First of all, for every set of such permissions a separate activity was added where the user can grant them to an app. Currently, there are 3 different activities responsible for granting such permissions (an example is given in Fig. 1b): to grant the *usage access* (`PACKAGE_USAGE_STATS`), *draw over other apps* (`SYSTEM_ALERT_WINDOW`), and *modify system settings* (`WRITE_SETTINGS`) privileges. Interestingly, these activities are accessed through different configuration screens: the first one is located under the "Security" settings, while the last two are on the "Configure apps" screen. This design decision is inconvenient for the users who must look in different locations to grant these permissions. Moreover, internally these activities are represented as 3 different classes with the corresponding permissions hardcoded within each class. Thus, if any new *appop* permission appears in the future, this will require the OS developers to add a new class processing this permission. In our study, we have also discovered one particular permission `CHANGE_NETWORK_STATE`, which in Android 6.0 were an *appop* permission. However, with the release 6.0.1 (i.e., still within API 23) its protection level was relaxed to *normal*.

Development permissions. These permissions (with `FLAG_DEVELOPMENT` set), although being of the *signature* protection level, can be granted to third-party applications by using the `pm grant` shell command. While the code for granting and revoking *development* permissions in Android 6.0 was merged with the one handling *runtime* permissions, these groups are quite different. First, *development* permissions are granted simultaneously to all system users, while *runtime* – only to the current user. Second, they are not displayed in the user interface as *runtime* permissions.

Pre-23 permissions. The permissions with `FLAG_PRE23` set are automatically granted to all legacy (whose target API level is below 23) applications requesting them.

Installer and verifier permissions. These *signature* permissions are automatically granted to the apps marked as required installer and verifier.

5) The deprecated *signature|system* protection level. Although the *signature|system* protection level is now deprecated, Fig. 3a and Fig. 3b show that there are still many permissions using this deprecated value. What is even

more confusing, 9 new permissions of this level appeared in API 23. We attribute this inconsistency to the lack of communication among the groups of developers responsible for different modules. We have developed and submitted to AOSP [1] patches to fix these issues. Currently, *out of 9 submitted patches, 2 patches were merged into the master branch, while 3 were verified and 5 were code-reviewed.*

6) Some dangerous permissions are now normal. In Android 6.0 the amount of *dangerous* permissions was considerably reduced. For 22 *dangerous* permissions the protection level was lowered to *normal*. Thus, the users now do not have any control over functionality protected with these permissions: *normal* permissions are not displayed and are automatically granted upon the installation. At runtime, a user can neither check them nor revoke. For instance, the `INTERNET` permission controlling the access of apps to the Internet, which was widely used by malware [55] especially in combination with other permissions [32], is now granted automatically.

From the security perspective, this is one of the most controversial changes, because many permissions regarded before as sensitive are now granted automatically. The fact that 22 permissions (including, e.g., `NFC`, `BLUETOOTH`, `WRITE_PROFILE`, `MANAGE_ACCOUNTS`) have been demoted in the security level emphasizes that the Android security architecture is far from being stable.

6.2 Interesting findings

1) Protection level flags. Developers cannot use protection level flags in their third-party apps. An application containing permission declaration with protection level flags will not pass validation checks during the compilation. The developers may only select one of the four main protection levels for their custom permissions: (*normal*, *dangerous*, *signature* and *signature|system*). At the same time, the validation check is performed only during application compilation. During installation of an app similar checks are not fired, and it is possible to add a protection level flag through app repackaging, e.g., using apktool[7]. Clearly, the checks in IDEs should conform to the new permission specifications, i.e., the *signature|system* protection flag should be removed, and there should be a possibility for third-party application developers to assign protection level flags to their custom permissions.

In Android 6.0 the protection level flag `FLAG_PREINSTALLED` was added. Previously, all *signature* permissions were divided into *privileged*, which could be obtained only if a system app was installed in the special folder, and others, which could be obtained by apps signed with the same certificate. `FLAG_PREINSTALLED` relaxes this strict division, and permits all system apps to receive automatically the permissions with this flag set.

2) Additional flags. Currently there are no restrictions for a third-party developer on assigning additional flags to custom permissions. For instance, it is possible to declare a permission with `FLAG_COSTS_MONEY` set. As a result, on older systems you will see the corresponding permission accompanied with a special

[7] `https://ibotpeaches.github.io/Apktool/`

coins icon. Similarly, the usage of `FLAG_HIDDEN` is also not restricted. This may be used by a developer to conceal a permission from the list of app's *dangerous* permissions. While we cannot say if this functionality can be used with malicious purposes, these edge cases violate the principle of least privilege.

Moreover, as mentioned in Sec. 4.2, 2 out of 3 flags can be set by a developer, while the third flag `FLAG_INSTALLED` can be installed only by the operating system. Such behavior is considered as security anti-pattern, when publicly accessible data is combined with private information.

3) Hard-coded screens for granting permissions. Every permission group defined in the core `AndroidManifest.xml` file has its own screen, where a user grants and revokes permissions assigned to this group (see Fig. 1c for the entry points to these screens). At the same time, permission groups defined in the system or third-party packages do not have dedicated screens. The "Additional permissions" screen collects all of them. There is no separation between groups and single permissions on this screen. E.g., Fig. 1d shows that the permission group (*test permission group*) and the single permission (*test single permission*) are listed on the same screen along with other groups defined in system packages. As we mentioned, the groups and single permissions will be displayed on this screen only if the corresponding package targets API 23.

4) Permission groups. We mentioned that there is no restriction on adding custom permissions to the system permission groups. If a custom permission has the *dangerous* protection level, then, when an app requests this permission at runtime, it is also granted with all permissions from the same group. At the same time, if the protection level of a custom permission is not *dangerous*, the remaining permissions from the group will not be automatically granted. Thus, to our point of view, there is no reason to group permissions beside those with the *dangerous* protection level. We analyzed system non-*dangerous* permissions to detect if there are any assigned to groups. For the API level 23 we found 6 such *package* permissions and 2 *core* permissions. For example, the `USE_FINGERPRINT` permission assigned to the `SENSORS` permission group has the *normal* protection level, while `ACCESS_IMS_CALL_SERVICE` belonging to the `PHONE` group has the *signature*|*system* level. We do not see reasons for this assignment and expect these issues to be fixed in the future Android releases.

5) Permission declaration duplicates. During our analysis we found that some permission declarations are duplicated even within AOSP. The most frequent duplicates are declarations of `INSTALL_SHORTCUT` and `UNINSTALL_SHORTCUT` permissions. These flags are declared both in the core and package manifest files. Before API 19 there were no declarations of these permissions on the core level, but due to a bug they were added to the core manifest file [6]. Interestingly, these permissions in the core and packages manifest files have different protection levels: *normal* in the former case and *dangerous* in the latter. Additionally, while exploring this issue, we discovered that in API 17 the declarations of two permissions (`SET_SCREEN_COMPATIBILITY` and `CHANGE_CONFIGURATION`) were duplicated even within the core file. This shows that some classes and configuration

files reached critical complexity within AOSP. It is necessary either to refactor them, or to use extensively static analyzers to prevent these inconsistencies.

7 Related Work

Studies in the literature investigated many aspects of the Android permission system [34, 37]. Indeed, the permission system is a cornerstone of the Android security model [31], while permission misuse is a great concern [27, 51], and permission request patterns in apps are widely used for pinpointing malicious or dubious behavior (e.g., [21, 28, 46, 52]). At the same time, Android developers require guidance for understanding permissions and using them correctly. For example, [35] and [26] looked at permission enforcement in Android and have shown that the principle of least privilege was often neglected by developers. Many studies looked into improving the permission system design and proposing more secure or more usable solutions (e.g., [29, 41, 45, 54]), while some researchers argued that finer granularity of permissions could be viable [38]. In absence of a reliable documentation from Google, researchers had also to provide a means of linking permissions to precise platform APIs that are protected with these permissions (a permission map) [23, 24, 26, 35, 40, 48]. Outside the Android platform, smartphone permission systems were explored in [22, 39, 47].

Wei et al. [49] have performed an early study of the permission system evolution in Android demonstrating that the permission system has become even more complex over time from the user's perspective (since its introduction in 2008 till the study publication in 2012). [49] revealed that the principle of least privilege was more and more violated with the time (the amount of overprivileged apps had consistently grown). Moreover, the permission system had become more complex: the total number of permissions had increased, and the amount of *dangerous* permissions had grown.

Au et al. [23] performed another longitudinal study of Android permissions with a focus on the sensitive API and permission changes spanning Android versions 2.2 up to Android 4.0. This study showed that the number of documented APIs requiring permissions had grown significantly in Android 4.0, and that many APIs changed their permission requirements over Android versions; this is also consistent with our own findings. The difference of our study with [23] is that we explore the changes in the permission system in the whole, while Au et al. concentrated on relations between permissions and API calls.

The studies by Wei et al. [49] and Au et al. [23] were reported in 2012. Thus, our study incrementally adds to theirs by surveying also more recent platform versions. To the best of our knowledge, the new Android permission system architecture, including runtime permissions, has not yet been extensively studied by the security research community. However, runtime permission requests were previously suggested by security researchers [50], and the effect of dynamic permission revocation on the Android apps has been empirically evaluated [33].

8 Conclusion

In this paper, we conducted a comprehensive study of the Android permission system. Driven by the aspiration to understand new runtime permissions, we discovered that the permission system has considerably evolved after its seminal description in [31]. To help security researchers and Android developers to understand better the new model and its implications, we presented an updated view on the permission system. Besides giving the overview and intrinsic details of the new design, we have shown its main changes during the last 6 years. At the individual permission level we discovered and reported many issues that have implications on the Android security state and research. These findings emphasise the dynamic complexity of the Android permission system that needs to be taken into account by the community.

References

1. Android Open Source Project, http://source.android.com/, accessed: 03/31/2016
2. Commit 2af5708: Add per uid control to app ops, https://android.googlesource.com/platform/frameworks/base/+/2af5708
3. Commit 2ca2c87: More adjustments to permissions, https://android.googlesource.com/platform/frameworks/base/+/2ca2c87
4. Commit 33f5ddd: Add permissions associated with app ops, https://android.googlesource.com/platform/frameworks/base/+/33f5ddd
5. Commit 3e7d977: Grant installer and verifier install permissions robustly, https://android.googlesource.com/platform/frameworks/base/+/3e7d977
6. Commit 4516798: Moving launcher permission to framework, https://android.googlesource.com/platform/frameworks/base/+/4516798
7. Commit 6d2c0e5: Remove not needed contacts related permissions, https://android.googlesource.com/platform/frameworks/base/+/6d2c0e5
8. Commit a90c8de: Add new "preinstalled" permission flag, https://android.googlesource.com/platform/frameworks/base/+/a90c8de
9. Commit ccbf84f: Some system apps are more system than others, https://android.googlesource.com/platform/frameworks/base/+/ccbf84f
10. Commit cfbfafe: Additional permissions aren't properly disabled after toggling them off, https://android.googlesource.com/platform/frameworks/base/+/cfbfafe
11. Commit de15eda: Scope WRITE_SETTINGS and SYSTEM_ALERT_WINDOW to an explicit toggle to enable in Settings, https://android.googlesource.com/platform/frameworks/base/+/de15eda
12. Commit e639da7: New development permissions, https://android.googlesource.com/platform/frameworks/base/+/e639da7
13. Dashboards, http://goo.gl/mFciT7, accessed: 03/31/2016
14. Google Says Android Has 1.4 Billion Active Users, http://goo.gl/aUuUNw, accessed: 03/31/2016
15. Microsoft Excel, https://play.google.com/store/apps/details?id=com.microsoft.office.excel, accessed: 03/31/2016

16. Microsoft PowerPoint, https://play.google.com/store/apps/details?id=com.microsoft.office.powerpoint, accessed: 03/31/2016
17. Not Just For Phones And Tablets: What Other Devices Run Android?, http://goo.gl/kQ4Pi8, accessed: 03/31/2016
18. Play Store Permissions Change Opens Door to Rogue Apps, http://goo.gl/nJCwoY, accessed: 03/31/2016
19. Requesting Permissions at Run Time, http://developer.android.com/training/permissions/requesting.html
20. Smartphone OS Market Share, 2015 Q2, http://goo.gl/WQwfZO, accessed: 03/31/2016
21. Arp, D., Speizenbarth, M., Hubner, M., Gascon, H., Rieck, K.: DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In: Proc. of NDSS (2014)
22. Au, K., Zhou, Y.F., Huang, Z., Gill, P., Lie, D.: Short Paper: A Look at Smartphone Permission Models. In: Proc. of SPSM (2011)
23. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: PScout: Analyzing the Android Permission Specification. In: Proc. of CCS (2012)
24. Backes, M., Bugiel, S., Derr, E., Weisgerber, S., McDaniel, P., Octeau, D.: On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In: Poster Session of IEEE EuroS&P (2016)
25. Barrera, D., Kayacik, H.G., van Oorschot, P.C., Somayaji, A.: A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android. In: Proc. of CCS (2010)
26. Bartel, A., Klein, J., Le Traon, Y., Monperrus, M.: Automatically Securing Permission-based Software by Reducing the Attack Surface: An Application to Android. In: Proc. of ASE (2012)
27. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Reza-Sadeghi, A., Shastry, B.: Towards Taming Privilege-Escalation Attacks on Android. In: Proc. of NDSS (2012)
28. Chen, K.Z., Johnson, N., D'Silva, V., Dai, S., MacNamara, K., Magrino, T., Wu, E., Rinard, M., Song, D.: Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In: Proc. of NDSS (2013)
29. Conti, M., Crispo, B., Fernandes, E., Zhauniarovich, Y.: CRêPE: A System for Enforcing Fine-Grained Context-Related Policies on Android. IEEE Transactions on Information Forensics and Security 7(5), 1426–1438 (2012)
30. Elenkov, N.: Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press, 1st edn. (2014)
31. Enck, W., Ongtang, M., McDaniel, P.: Understanding Android Security. IEEE Security and Privacy Magazine 7(1), 50–57 (2009)
32. Enck, W., Ongtang, M., McDaniel, P.: On Lightweight Mobile Phone Application Certification. In: Proc. of CCS (2009)
33. Fang, Z., Han, W., Li, D., Guo, Z., Guo, D., Wang, X.S., Qian, Z., Chen, H.: revDroid: Code Analysis of the Side Effects After Dynamic Permission Revocation of Android Apps. In: Proc. of ASIACCS (2016)
34. Fang, Z., Han, W., Li, Y.: Permission based Android Security: Issues and Countermeasures. Computers & Security 43 (2014)
35. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android Permissions Demystified. In: Proc. of CCS (2011)
36. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android Permissions: User Attention, Comprehension, and Behavior. In: Proc. of SOUPS (2012)
37. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and Enhancing Android's Permission System. In: Proc. of ESORICS (2013)

38. Fratantonio, Y., Bianchi, A., Robertson, W.K., Egele, M., Kruegel, C., Kirda, E., Vigna, G.: On the Security and Engineering Implications of Finer-Grained Access Controls for Android Developers and Users. In: Proc. of DIMVA (2015)
39. Gadyatskaya, O., Massacci, F., Zhauniarovich, Y.: Security in the Firefox OS and Tizen Mobile Platforms. IEEE Computer 47(6), 57–63 (2014)
40. Gibler, C., Crussell, J., Erickson, J., Chen, H.: AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In: Proc. of TRUST (2012)
41. Jeon, J., Micinski, K.K., Vaughan, J.A., Fogel, A., Reddy, N., Foster, J.S., Millstein, T.: Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In: Proc. of SPSM (2012)
42. Murphy, M.: Libraries and Dangerous Permissions, `https://goo.gl/NJAjMx`, accessed: 25/06/2016
43. Murphy, M.: Runtime Permissions, Files, and ACTION_SEND, `https://goo.gl/slhHoI`, accessed: 25/06/2016
44. Murphy, M.: You Cannot Hold Non-Existent Permissions, `https://goo.gl/nyDjUj`, accessed: 25/06/2016
45. Nauman, M., Khan, S., Zhang, X.: Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In: Proc. of ASIACCS (2010)
46. Pandita, R., Xiao, X., Wang, W., Enck, W., Xie, T.: WHYPER: Towards Automating Risk Assessment of Mobile Applications. In: Proc. of USENIX Security (2013)
47. Singh, K.: Practical Context-Aware Permission Control for Hybrid Mobile Applications. In: Proc. of RAID (2013)
48. Vidas, T., Christin, N., Cranor, L.F.: Curbing Android Permission Creep. In: Proc. of W2SP (2011)
49. Wei, X., Gomez, L., Neamtiu, I., Faloutsos, M.: Permission Evolution in the Android Ecosystem. In: Proc. of ACSAC (2012)
50. Wijesekera, P., Baokar, A., Hosseini, A., Egelman, S., Wagner, D., Beznosov, K.: Android Permissions Remystified: A Field Study on Contextual Integrity. In: Proc. of USENIX Security (2015)
51. Xing, L., Pan, X., Wang, R., Yuan, K., Wang, X.: Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating. In: Proc. of S&P (2014)
52. Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X.S., Zang, B.: Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In: Proc. of CCS (2013)
53. Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., Crispo, B., Massacci, F.: StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In: Proc. of CODASPY (2015)
54. Zhauniarovich, Y., Russello, G., Conti, M., Crispo, B., Fernandes, E.: MOSES: Supporting and Enforcing Security Profiles on Smartphones. IEEE Transactions on Dependable and Secure Computing 11(3), 211–223 (May 2014)
55. Zhou, Y., Jiang, X.: Dissecting Android Malware: Characterization and Evolution. In: Proc. of S&P (2012)