This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

Mules and Permission Laundering in Android: Dissecting Custom Permissions in the Wild

Julien Gamba, Álvaro Feal, Eduardo Blazquez, Vinuri Bandara, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez

Abstract—Android implements a permission system to regulate apps' access to system resources and sensitive user data. One salient feature of this system is its extensibility: apps can define their own custom permissions to expose features and data to other apps. However, little is known about how widespread the usage of custom permissions is, and what is the impact that these permissions can have on users' privacy and security. In this paper, we empirically study the usage of custom permissions at large scale, using a dataset of 2.2M pre-installed and app-store-downloaded apps. We find the usage of custom permissions to be widespread, and seemingly growing over time. Despite this prevalence, we find that custom permissions are virtually invisible to end users, and their purpose mostly undocumented. This lack of transparency can lead to serious security and privacy problems: we show that custom permissions can facilitate access to permission-protected system resources to apps that lack those permissions without user awareness. To detect this practice, we design and implement two static analysis tools, and highlight multiple concerning cases spotted in the wild. We conclude this study with a discussion of potential solutions to mitigate the privacy and security risks of custom permissions.

Index Terms—Android, Access control, Custom permissions, Mobile apps

1 Introduction

The Android operating system implements a permission-based mechanism to control how applications (apps) can access sensitive data and dangerous system features [49] such as user contacts, the camera, location sensors, or the system settings. Coupled with other protection mechanisms such as process sandboxing, the permission system empowers users to control what sensitive resources are accessible to which apps. The Android Open Source Project (AOSP) defines a standard set of permissions which are supported by most Android devices. Any Google-certified device [3], [4] must implement the whole set of AOSP permissions to guarantee their compatibility with the standard Android platform [5].

A decade of research in the use, enforcement, and usability of AOSP permissions has revealed severe privacy and security shortcomings inherent to the Android permission model [71], [90], [73], [61], [86], [84], [65], [68]. However, the research literature overlooked a key feature of Android's permission model: its *extensibility*. By design, the Android framework allows any app developer to share features implemented in their software with other apps in a *"controlled"* way by defining *custom permissions* [47]. Therefore, custom permissions allow extending the capabilities offered by the Android OS and facilitate the flourishment of an open software ecosystem in which apps (and third-party libraries or SDKs) can share data and components with other developers. However, custom permissions pose potential security and privacy risks as they can be (ab)used—intentionally or by mistake—to circumvent the standard permission system and provide backdoored access to privileged data and system features to apps that are otherwise not permitted to do so, in a way akin to how covert- and side-channels operate [84].

The control and transparency mechanisms implemented by the Android operating system are insufficient to protect users from abusive or insecure implementations of custom permissions. Even identifying the party responsible for their definition and their purpose can become a daunting task. Google recommends using the reverse domain name as the prefix of such permissions, and supplying a description of the custom functionality or data protected by the permission [47], [6], but, in practice, there is no enforcement of such recommendations [75]. Consequently, it is not possible to automatically know what precise function or resource is protected by a custom permission, and how they are being integrated and used across Android apps. This lack of control and transparency also manifests at installation time, which translates into profound implications in terms of user awareness and control: unlike official AOSP permissions custom permissions are not listed in the app stores, and end-users cannot grant or deny apps access to them at runtime unless the developer willingly defines them with a `dangerous` protection level.

Despite these risks, the research literature on custom permissions is significantly narrow. Prior work performed a high-level analysis of the prevalence of custom permissions in pre-installed apps [75], while others demonstrated, using

- *Julien Gamba, Álvaro Feal and Vinuri Bandara are with the IMDEA Networks Institute and the Universidad Carlos III de Madrid, Spain.*
- *Eduardo Blazquez and Juan Tapiador are with the Universidad Carlos III de Madrid, Spain.*
- *Abbas Razaghpanah is with ThousandEyes/Cisco, San Francisco, CA, USA; and ICSI, Berkeley, CA, USA.*
- *Narseo Vallina-Rodriguez is with the IMDEA Network Institute, Madrid, Spain; and AppCensus Inc.*

Manuscript received May 2nd, 2022

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

proof-of-concept implementations, how custom permissions can enable permission re-delegation and confused deputy attacks [63], [64], [87], [79]. Yet, our understanding of the Android custom permissions landscape has remained low, particularly in terms of their prevalence, usage, and potential misuse. In fact, the state-of-the-art lacks app analysis tools capable of capturing and analyzing the risks of custom permisisons due to their asynchronous nature.

In order to fill this knowledge gap, we study a dataset of 52,468 unique custom permissions defined by publicly-available and pre-installed apps. This is the largest dataset of custom permissions collected to date and gives us an unprecedented and global view of this ecosystem (§4). Using this dataset, we make the following key contributions:

- We present the first longitudinal and large-scale measurement of the usage of custom permissions in the Android ecosystem (§5). We find that both pre-installed and public apps both define and request a large number of custom permissions. Namely, 58% and 67% of pre-installed and public apps request at least one, and 26% and 4% define at least one, respectively.
- We measure whether developers defining custom permissions comply with Google's naming and transparency recommendations, finding widespread violations. Specifically, 45% of declarations do not follow naming recommendations. For example, we find 722 custom permissions with the `android.permission` prefix, which is explicitly forbidden by the Android Compatibility Definition Document (CDD). Moreover, there is no enforced mechanism by which developers have to report what a given custom permission enables or is used for. While there is a `description` tag to describe the purpose and functionality of the custom permission, its usage is optional and we find that it is rarely used by developers, being missing in 75% of the cases.
- The lack of transparency in custom permisisons is aggravated by the lack of analysis tools to trace and understand the type of data or capability that a given custom permission protects. To fill this methodological and tooling gap, we present a novel method to triage apps that are potentially misusing custom permissions to access personal data, or perform other actions potentially detrimental to users' privacy and security (§7). Our method relies on two purpose-specific tools: (1) `permissionTracer`, a tool that reports potentially-dangerous custom permissions and detects potential cases of a privilege escalation attack in which an attacker can access permission-protected information using custom permissions; and (2) `permissionTainter`, a static taint analysis tool that inspects the DEX code of apps that define custom permissions, to identify potential privacy leaks due to those permissions. Equiped with these tools, we identify several instances of potentially harmful and insecure implementations that can expose sensitive data such as the location, Wi-Fi MAC address, or contacts without requesting the corresponding AOSP permission.
- We conduct a small-scale survey of app developers who defined some of these custom permissions in order to understand their use case and rationale (§7). Our findings suggest that most developers lack a clear understanding of their purpose and functioning. As a result, custom permissions are often used due to poor software development practices or because they are required to define them in order to integrate third-party SDKs.

These four contributions offer a unique picture of the custom permission landscape and introduce new methods and tools for assessing their security and privacy risks. We conclude this paper with a constructive discussion on potential solutions for the accountability and transparency issues of Android custom permissions. To foster further researc in this domain and raise awareness about the risks of custom permissions, we make our dataset of custom permissions available to the research community [21], [20].

Responsible disclosure. During the course of this study, we identified vulnerabilities in Android apps currently installed on user devices. To minimize negative consequences for users, we have responsibly disclosed our results to Google in December 2020, including several examples of apps that expose private data without user consent via custom permissions. Google representatives acknowledged the issue, but considered it to be a consequence of the openness of the Android platform, and therefore difficult to solve (and monitor) without hindering the possibility for developers to create custom permissions.

2 The Android permission system

This section provides essential background knowledge about the Android permission system and its extensibility through custom permissions. We refer the reader to Google's official documentation for general details on the Android permission system [49].

2.1 Permission model

Android's security model leverages some of the security features offered by the Linux kernel, including user isolation. In Android, each app runs under a unique user ID (UID), belongs to a group whose group ID (GID) is the same as the app's UID, and is given a dedicated access-protected data directory. User apps are sandboxed at the process and file system level, thus preventing them from arbitrarily interacting with each other. To access sensitive user data (e.g., text messages or contacts), device features (e.g., camera or GPS), and OS services (e.g., system settings), apps must request and be granted specific *permissions*.

Each permission in Android is assigned a user group with a distinct GID. The kernel manages access to resources such as regular files, devices, and local sockets, based on an apps' group membership by way of its UID and associated GIDs. When an app is granted a permission by the framework (action performed by Android's `ActivityManager`), the UID assigned to the app becomes a member of the group assigned to that permission, thus effectively granting the app access to the resources it protects.

It is also important to note that all Android apps are cryptographically signed with a digital certificate [55]. In Android, there are special mechanisms in place for apps that are signed with the same certificate to share data more easily, as they are meant to belong to the same "developer". Namely, apps signed with the same certificate can use the

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

sharedUserId attribute in their manifest to request the system to run with the same UID. This means that such apps can run in the same process and share access to the same system resources; i.e., any permission granted to one of the apps will be granted to all other apps signed with the same cerificate. This feature was deprecated in API level 29 [11]. We note, however, that the certificate is only a weak attribution signal, as Android apps rely on self-signed certificates and, as such, the information they contain cannot be fully trusted [77].

2.2 Requesting permissions

Apps must include the <uses-permission> tag in the Android Manifest file for requesting permissions [49]. Each Android permission has an associated *protection level* that relates to its implied potential risk. This, however, affects the procedure that the operating system follows to determine whether or not to grant a given permission to a requesting app:

- Permissions with the normal protection level are considered not to pose much risk to the user's privacy or the device's security, and are automatically granted at installation time.
- Permissions with a signature protection level will also be granted by the system at installation time, but only if the app requesting the permission is signed with the same certificate as the app defining it.
- Finally, dangerous permissions protect resources that are considered sensitive (e.g., the device's location) and therefore require explicit user approval. dangerous permissions are granted at runtime since Android 6.

Permission can also be part of a *permission group*, which gather together permissions that refer to the same part of the system (e.g., the READ_SMS and WRITE_SMS permissions are both in the SMS group). Permission requests are handled at the group level, even if each single permission definition appears in the manifest.

2.3 AOSP permissions

The Android Open Source Project defines a set of standard permissions that must be supported by Google-certified Android devices. These aim to define the standard way of accessing the most common resources across different devices, such as obtaining the location or sending text messages. The labels for these permissions begin with the android.permission prefix. For instance, android.permission.SEND_SMS is the standard AOSP permission for sending text messages.

The permission system has evolved and increased over time as illustrated in Figure 1 as a result of Google adding new features for device manufacturers or developers, or improving the security and privacy guarantees of the system. The number of AOSP permissions has grown from 114 in Android 1.6 (API level 4, released in 2009) to 689 in Android 12 (API level 31, released in 2021). Not all of these permissions are supposed to be available to all developers though: some are marked as *"Not for use by third-party applications"* in the AOSP source code (e.g., the READ_LOGS permission which allow an app to get access to the system

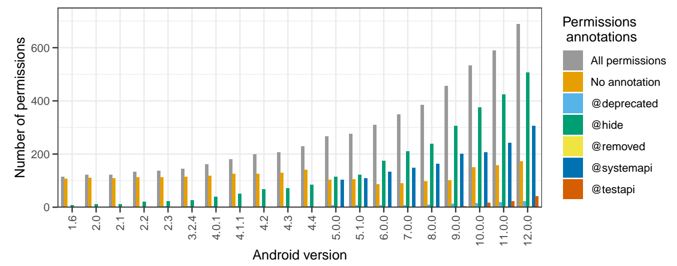


Fig. 1: Evolution of the number of AOSP permissions per Android release observed by parsing the manifest file of the open-source framework app, which defines those default permissions for the system.

log files). In fact, out of the 689 official permissions defined at API level 31, 305 permissions (44% of the total) have the @SystemAPI annotation, which indicates that they are reserved for system processes.

2.4 Custom permissions

Android allows developers to define (or expose) their own *custom* permissions to enable controlled access to their own components and features. This facilitates *"regulated"* programmatic inter-app communication and data sharing, despite each app running with a different UID, as illustrated in Figure 2. Apps can define their custom permissions in their manifest file by using the <permission> tag [7]. Apps declaring (or requesting) access to custom permissions must also do so in the manifest file, just like they do for regular AOSP permissions. However, for apps published on the Google Play market, the custom permissions requested by a given app is not rendered in its public market profile, so users are unaware of their presence at the time of installing them from the markets. Once an app requests access to a custom permission, it can interact with the protected component, for instance by sending an intent [57] or by instantiating the component directly (e.g., for a protected activity). By default, access to custom permissions is regulated by the OS package manager, but the app defining them can implement further access controls to only grant access to authorized apps, regardless of the protection level of the permission, by calling checkPermission, enforcePermission or one of their variants [56]. The ways in which other apps can access an app component depend on whether it is exported (either through the android:exported attribute or if it contains any Intent-Filter) and on the protection level of the protecting permission.

2.5 Naming conventions for custom permissions

The Android operating system does not impose any restriction on custom permission names or the features and data they can enable. However, Google recommends using the app's package name as the prefix for the custom permissions that it defines (e.g., an app with the package name com.foo should name their custom permissions com.foo.MY_PERM), which itself should use a "reverse-domain-style name", to ensure package and permission name uniqueness [47]. Google also recommends adding a description of

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

Fig. 2: Example of an app defining a custom permission and protecting a service with it. Only App_3, which requests the permission, can interact with the service exposed by App_1.

the purpose of their permissions to "explain the permission to the user" [36] when defined in the Android Manifest file. However, no active policy enforcement is applied [75]. Moreover, Software Development Kits (SDK) embedded on Android apps may also define and expose custom permissions with the collaboration of the developer, in which case the permissions they request or define will be merged in the manifest of the host app [33] (e.g., an SDK from `sdk.com` could define the `com.sdk.MY_PERM` permission). The presence of SDK-defined custom permissions adds another layer of complexity to the analysis and attribution of custom permissions to the responsible party.

3 Related Work

Previous work on Android's permission system has focused on the usage and abuse of AOSP permissions [86], [61], over-privileged apps [71], [67], [82], detecting vulnerabilities and weaknesses in the permission system [74], [66], [76], [85], [84], [65] and assessed the efficacy and transparency of Android's permission model to empower users [73], [72], [78], [90]. Multiple tools were also created to study AOSP permissions. Felt et al. presented Stowaway, the first dynamic analysis tool to determine if all permissions requested by a given app are actually used in runtime [71]. The authors ran Stowaway on 900 Android apps and found that around 35% of them asked for unnecessary permissions (i.e., they were not used on the app's code). They demonstrated that over-privileged apps are typically the result of developer errors (e.g., legacy code, or copied-and-pasted code). Au et al. proposed PScout, a static analysis tool to automatically infer the specification of the permission system from Android 2.2 to Android 4 [61]. Their main objective was to determine if, given the large number of permissions offered by the OS (79 at the time of publication), there was any overlap in the set of protected APIs for a given pair of permissions, and found only one such pair. The authors also noted the presence of undocumented APIs and permissions, but show that such APIs are rarely used by third-party app developers.

Backes et al. addressed the same problem and built a static runtime model of the Android permission framework [62] to create a more complete and recent mapping of API calls to permissions. The authors studied permission locality (i.e., whether permissions are enforced only by one particular service). They showed that 20% of the analyzed permissions are checked by more than one single class, making enforcement of permissions a more complex task, as it violates the principle of separation of duties (i.e., in this case, multiple AOSP components are responsible for enforcing the same permissions). Finally, Reardon et al. revealed how app developers exploited covert- or side-channels to gain access to permission-protected data, thus circumventing the Android permission model. For example, developers gathered the MAC address of the device without holding the otherwise-required permission by calling `ioctl()` [84].

TABLE 1: Number of unique apps (by their MD5 hash) and custom permissions per source. We merge AndroZoo apps with their actual market of origin if we actively crawl said market (e.g., AndroZoo apps fetched from the Play Store are considered in the Google Play set). Otherwise, they are considered in the "AndroZoo" category.

Origin	Number of apps	Number of permissions		
		requested	defined	all
Google Play	638,758	19,464	13,626	22,010
Tencent	94,443	11,610	7,013	12,591
APKMonk	23,774	1,037	402	1,108
Xiaomi Mi	21,613	6,381	3,852	6,838
Baidu	11,522	3,172	1,810	3,358
APK Mirror	9,696	2,106	852	2,246
Huawei	6,655	3,613	2,227	3,895
Qihoo 360	4,321	3,092	1,524	3,251
AndroZoo	217,639	9,814	6,195	10,660
Pre-installed	1,247,447	16,886	14,912	19,312
Total	**2,234,506**	**46,556**	**37,743**	**52,468**

3.1 Custom permission analysis

The research literature on custom permissions is very narrow. Tuncay et al. [87] revealed vulnerabilities on Android's permission system related to custom permissions. They described a custom permission upgrade attack that exploits the permission groups to be able to enable any dangerous permission without user awareness and approval. They also discussed a confused deputy attack that exploits the lack of enforcement on naming conventions to access signature custom permissions with an app that is not signed with the same certificate as the defining app. Both attacks were acknowledged and fixed by Google. The lack of enforcement on naming conventions has both transparency and security implications. Bagheri et al. [63], [64] formally validated the Android permission model and showed that the lack of compliance with the naming conventions for custom permissions allows an attacker to access components protected by a custom permission in the victim app, in a way akin of the confused deputy attack described by Tuncay et al..

Li et al. show how custom permissions can be used to gain access to APIs otherwise protected by AOSP permissions [79]. The authors develop CuPerFuzzer, an automatic fuzzing tool that they use against the Android OS. This tool allowed them to discover four design shortcomings of the permission system, which were reported to Google and fixed by the Android security team. However some of these attacks need user interaction to be carried out, which renders them less practical. Their attack has since been fixed in Android 10.

Finally, in our prior work, we performed a preliminary analysis on the Android supply chain [75] and identified a large number of custom permissions in pre-installed apps, many embedded even in core Android components. Our preliminary study, however, did not perform any systematic analysis of their associated privacy and security risks, nor about its usage by regular apps.

4 Data collection

For this paper, we gathered a large-scale dataset of both user-installed and pre-installed Android apps between 2019

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

and 2022 as shown in Table 1 that offers a holistic perspective of apps exposing and requesting custom permissions.

Public app stores. We implemented a purpose-built crawler to download apps and their associated metadata from several public app stores at scale: Google's Play Store [25], Tencent [44], APKMonk [13], Xiaomi's Mi Store [45], Baidu [14], APK Mirror [12], Huawei [27], and Qihoo 360 [40]. We chose these app stores for their popularity, thus giving us access to a representative picture of the Android ecosystem including and beyond the Play Store [88]. We complement this corpus with apps collected by the AndroZoo project [50].

Pre-installed apps. We rely on our dataset of pre-installed apps that we collected via crowdsourcing mechanisms using Firmware Scanner, a purpose-built app available on the Play Store [22]. The dataset contains metadata about the devices (e.g., brand, model, and country) in which the apps come pre-installed. We refer the reader to our paper [75] for a detailed description of the operation of Firmware Scanner. The dataset contains 1,247,447 apps collected from 58,540 users, representing 17,973 unique device models associated with 783 Original Equipment Manufacturers (OEMs). To account for different apps sharing the same package name but potentially manipulated by different developers—a common occurrence in pre-installed applications, where core Android components can be modified by the vendor—, we identify unique apps by their package name and certificate.

4.1 Methodology for extracting custom permissions

We consider any permission to be custom if it never was in the official list of AOSP permission for any Android release. We therefore start by extracting the official AOSP permissions across Android releases by parsing the manifest of the open-source AOSP framework app [10]. Then, to extract custom permissions, we parse the apps' manifests and extract `<permission>` tags for defined permissions, and both `<uses-permission>` tags and the `android:permission` attributes of permissions protecting apps' components for requested custom permissions. Using this approach, we obtain 257,710 unique custom permissions, either defined or requested ones. Alongside the permission name, we also extract metadata related to the app that defined or requests it (e.g., app's package name and signing certificate), and information about the permission itself (e.g., description field and protection level) to further study the adoption of naming conventions, and developers' willingness to document their custom permissions.

Attribution. We leverage Google's naming recommendation as a proxy to identify the party responsible for the definition of custom permissions. For example, `com.foo.PERMISSION` has the second-level domain `foo.com`, which should identify the author of the custom permission. However, as mentioned earlier, developers do not necessarily abide by this convention. Relying on extra signals such as the app's signing certificate does not solve this issue, as applications in Android use self-signed certificates, and previous work showed the existence of applications purposefully using false information in their certificate to impersonate other companies [75], [77]. Due to the lack of robust mechanisms to do sound attribution of apps and custom permissions, we rely on the naming convention as the only way to potentially understand who defined a given permission. However, when available, we rely on online documentation as a reliable source for (1) attributing permissions to app developers or SDKs; and (2) inferring what service or data the permission is protecting. In some cases, we manually inspect the package name of the app and the signing certificate to enhance our attribution process but the scale of the dataset prevents us from performing this process for every single app.

Push notification services. Push notifications are messages displayed to the user, either from a local app, or from a remote server even when their app is not running on the device. Developers include a receiver in their app to receive the notifications, which they protect with a custom permission to prevent other apps from intercepting the messages. We identified several push notification services from companies such as Xiaomi [46], Amazon [1], and others [31], [26], [52], [28], [15], [24]. Due to their widespread use and its supposedly harmless nature, we exclude 205,242 permissions associated with such services for the rest of the paper. However, we note that it is technically possible for a malicious app to create a permission resembling the syntax of a push notification service for harmful purposes. After applying this filter, we consider 52,468 custom permission names for this paper, both requested and defined.

4.2 Ethical considerations

Our data collection relied on real users that installed Firmware Scanner on their devices. We follow the principles of informed consent [69] and avoid the collection of any personal or sensitive data. The app does collect some metadata about the device (e.g., its model and build fingerprint) along with some data about the pre-installed applications (extracted from the Package Manager), network operator (MNO), and user (the timezone, and the MCC and MNC codes from their SIM card, if available). Additionally, using the developer contact details available on Google Play, we also survey app developers making use of custom permissions to better understand their rationale and the reasons why they include them (§7). We treat this as sensitive data since it might have unexpected consequences, e.g., for their current and future employment. We therefore only report statistical and anonymized data, and do not store any information that could be used to identify a particular developer or company. In both cases, we consulted our data collection protocols with IMDEA Networks' Data Protection Officer (DPO) and received approval from our institutional ethical review board to conduct this survey.

5 Prevalence of Custom Permissions

Table 1 preliminary results suggest that there is a significant usage of custom permissions, both requested and defined, regardless of the type of app or its origin. However, these numbers by themselves do not completely convey the scale and complexity of the custom permissions ecosystem, especially the number of actors involved. This section measures how widely defined and requested custom permissions are at the application- and market-level.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

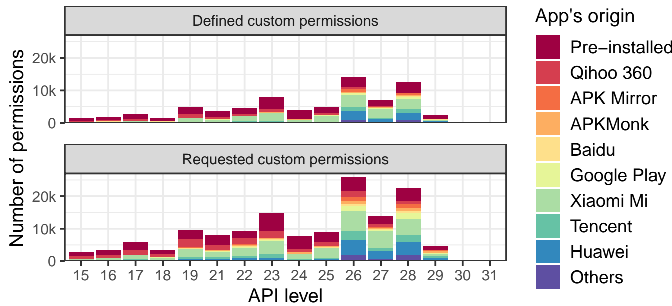


Fig. 3: Number of custom permissions requested or defined per target API level, broken down by the origin of the app. For clarity reasons, we exclude apps that target an API level lower than 15 as less than 0.1% of current Android devices run such an old version [9].

5.1 Definition of custom permissions

Figure 3 shows the increasing number of defined custom permission per API level targeted by the app, i.e., as new Android versions get released. We note that the low number of permissions for API level 30 and up is due to the fact that our dataset only contains 85 applications targeting such API levels, all origins included. We find that the proportion of apps defining custom permissions is much lower than the proportion of apps requesting them: only 4% of apps available on app stores define at least one custom permission versus 26% of pre-installed apps. In fact, the Android Open Source Project allows OEMs to define and expose their own services to other apps through custom permissions.

The reasons why custom permissions are declared are diverse. By comparing the device fingerprint reported by Firmware Scanner with the prefixes of the custom permissions exposed by pre-installed apps, we could label 63% as OEM-defined. While most OEMs define custom permissions, Samsung, Huawei and Amazon devices tend to define more than the average. In fact, just Samsung defines 4,822 custom permissions, 109 of which are related to Samsung's Knox framework, a proprietary security framework that offers features like access control, mobile device management, and VPN capabilities [43], [35], [37]. Anecdotically, we found a Samsung device that defines as many as 664 custom permissions. Many OEM permissions are defined by core Android components (e.g., the default dialer app (`com.android.phone`) customized by OEMs to add their own features and services. Figure 4 renders a boxplot of the number of custom permissions defined by such core apps. This shows the high number of potential vulnerable features made available by privileged and critical pre-installed apps to other applications, including applications distributed through Android stores.

Yet, not only OEMs define custom permissions. By reasoning about their prefix, 34% custom permissions are related to companies offering third-party analytics and advertising SDKs [80], [83], [70] (e.g.,Baidu, AppsFlyer) or social networks (e.g., Facebook, Twitter). For example, according to their official documentation, the permission `com.twitter.android.permission.AUTH_APP` is used for allowing users of a given app to log in through Twitter, and `com.baidu.permission.BAIDU_LOCATION_SERVICE` is related to Baidu's map services. Another interesting set of custom permissions are 6 permisisons for enabling IoT platform integration. For example, the permission `amazon.speech.permission.SEND_DATA_TO_ALEXA` relates to Alexa devices [30], while 51 are related to Google's Android for cars services [17], including accessing car-specific information such as the speed of the vehicle (`android.car.permission.CAR_SPEED`) or control of the lights (`android.car.permission.CONTROL_CAR_INTERIOR_LIGHTS`).

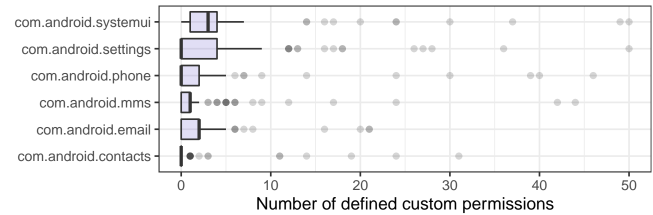


Fig. 4: Number of custom permissions defined by core Android components. Note that we do not include the `android` app in this plot for readability reasons.

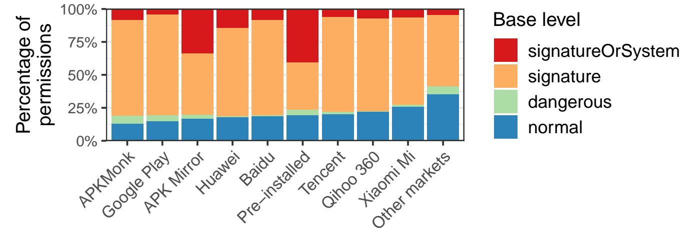


Fig. 5: Base protection level usage per origin of the app for defined custom permissions.

Protection level analysis

As with regular AOSP permissions, custom permissions can set different protection levels to regulate its access. Figure 5 shows the protection level of defined custom permissions per app type. This figure shows that 39% of the custom permissions are defined with a `signature` protection level. When considering exposed custom permissions with a `signatureOrSystem` protection level, this proportion rises up to 86%. This means that the majority of custom permissions will only be granted to apps that share a signing certificate with the declaring app as we will study at the end of this section. [1]

More concerning is the fact that 11% of the permissions are defined with the `normal` protection level. Motorola, HTC and Xiaomi define a total of 170, 193, and 269 custom permissions with the `normal` protection level. For Samsung, this number goes as high as 867 custom permissions. Therefore, any app installed on the same device will automatically get granted these permissions at installation time unless the developer defining the permission implements other access control mechanisms programmatically (e.g., by checking the

1. We note that the protection level `signatureOrSystem` is deprecated since API level 23 (Android 6.0) [41] and it is semantically equivalent to the `signature` base protection type with the `privileged` flag, which allows an app installed on the system partition to be automatically granted the permission when requested [18].

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

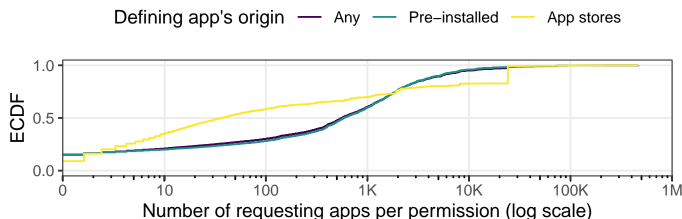


Fig. 6: Number of apps requesting custom permissions in our dataset, broken down by the origin of the defining app

package name of the calling app). Unfortunately, the lack of public information about the actual purpose of these custom permissions (or the type of data or service that they protect) and tools for automatically analyzing their risks has historically prevented us from assessing wheter sensitive data is left unprotected, as we will demonstrate in Section 7.

5.2 Requests of custom permissions

Figure 3 shows the number of requested custom permissions per target API level and origin of the app. In general, 30% of apps published in public app markets request at least one custom permission but this number is significantly higher (62%) for pre-installed apps. When ranking them by their prefix and popularity—which we define as the number of apps requesting them—we can observe clusters of popular custom permissions. Table 6 in the appendix shows the top 20 most requested custom permissions, along to their potential creator which we infer from the `Subject` field of the app signing certificate and its prefix. As we can see, Google Mobile Services (GMS) permissions are requested by more than 10,000 apps and they enable Google-related functionalities related to in-app purchases [54] (`com.android.vending.BILLING`), the Play Install Referrer Library [51], [53] (`com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE`) and Google Sign-In [48] (`com.google.android.gms.auth.api.signin.permission.REVOCATION_NOTIFICATION`). Samsung permissions are also amongst the most widely requested by app developers.

OEM-specific custom permissions

Figure 6 shows that custom permissions defined by pre-installed apps are likely to be requested by more apps than those defined by publicly available apps. Specifically, the median number of requesting apps per permission is of 587 and 36 for pre-installed and publicly available apps, respectively. This confirms the importance of inspecting potential vulnerabilities on pre-installed apps, as we will further discuss in Section 7.

Figure 7 provides a more detailed perspective on how the OEM-specific permissions for the top-10 Android OEMs are requested by publicly available apps.[2] For completeness, we include Google Mobile Services permissions exposed by pre-installed apps on 87% of the devices in our dataset. We group the remaining vendors under the “Others” label. We can infer two things from this figure: (i) a large number of permissions exposed by pre-installed apps are primarily requested by other pre-installed applications, which could indicate the existence of partnerships between actors of the supply chain of Android devices; and (ii) apps from all app stores do request OEM-defined permissions. Specifically, a total of 43,517 applications in our dataset request Samsung Knox permissions, but 98% of them are other apps pre-loaded on Samsung devices. Those apps distributed through Google Play and requesting Knox services are mostly professional applications like Cisco Webex, and MDM solutions. This confirms that the important role of pre-installed apps in the development of Android applications and the need for assessing their security and privacy risks.

2. We infer OEM popularity by the number of users with devices of a given OEM in our dataset. Yet, our top-10 vendors correlate to publicly available Android market shares [32].

TABLE 2: Most popular second level domains for custom permissions defined or requested by apps on public stores.

Origin	Most popular SLD requested perms	Most popular SLD defined perms
Google Play	google.com	google.com
Tencent	google.com	tencent.com
APKMonk	google.com	sina.com
Xiaomi Mi	permission.android	tencent.com
Baidu	permission.android	lechuan.com
APK Mirror	google.com	google.com
Huawei	permission.android	huawei.com
Qihoo 360	permission.android	tencent.com
Others	permission.android	permission.android

Market-level differences

At the market-level, we see that apps published in Xiaomi Mi, Tencent and Huawei app markets tend to request more custom permissions than apps published in Google Play. We note that some markets are more recent than others. For instance, the Huawei app store was only launched globally in 2018 [29]. Their short age might explain why the usage of custom permissions in Huawei's market is higher for higher API levels. Nevertheless, the declaration of custom permissions in Android apps grows with new Android releases: the median number of requested custom permissions between API levels 15 and 25 (both included) is 5,303.5 per API level, while for API levels 26 to 31 (478,244 of all apps in our dataset), the median rises up to 9,550 requested custom permissions per API level.

Table 2 shows the most requested permission (grouped by their SLDs) for apps publicly available on public app stores. These figures stress the importance of Google permissions in Android app development, being the most popular requested permissions in half of the markets we cover. Google apps present on Google Play (including very popular ones such as YouTube, GMail or the Google Play Services app which is also pre-installed on any Google-certified device) define as many as 183 custom permissions with the `com.google` prefix. We find that the `android.permission` prefix is the most requested permission in app stores from China. For instance, the `android.permission.DOWNLOAD_WITHOUT_NOTIFICATION` permission (which is not part of AOSP) is requested by 5,757 applications on the Baidu app store alone. Permissions seemingly from Google (i.e., in the `google.com` SLD

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

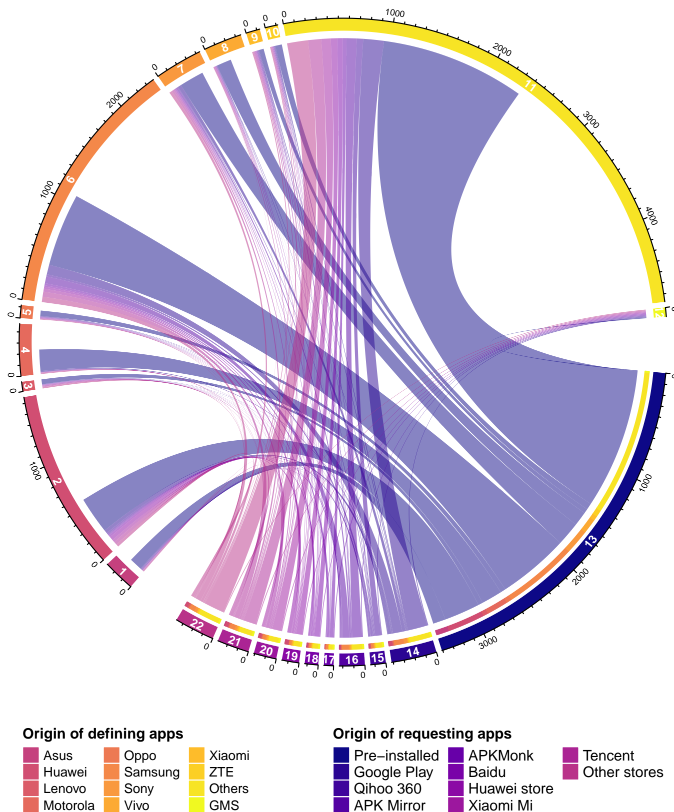


Fig. 7: Number of requested permissions defined by pre-installed apps, broken down by the origin of the requesting app (left side) and OEM (right side).

group) are still in the most popular ones in Chinese markets, but followed by well-known Chinese companies such as Tencent or Sina Corporation. This shows a geographical divide between Chinese app stores and Google Play.

Signature permissions

One final aspect to consider is the link between exposed and requested custom permissions with signature level, as they can be automatically granted during installation time. When a custom permission has a `signature` or `signatureOrSystem` protection level, we check the certificate(s) of both the defining and requesting app, and identify cases where both apps have at least one certificate in common. This approach allows us to reproduce the behavior of the Android OS at granting these permissions.

We focus on custom permissions that are declared by pre-installed apps, as those apps are inherently more trusted by the operating system [75]. We find that custom permissions declared by pre-installed apps are mostly requested (and, in this case, granted) to other pre-installed apps: out of the 586,354 apps that would be granted `signature` or `signatureOrSystem` permissions, 99.9% of them are pre-installed. We find 13,717 apps (2.3% of the total) on public markets that would also be granted such permissions automatically. In particular, we find that some Facebook apps—including the official Facebook app (`com.facebook.katana`) and Facebook Messenger (`com.facebook.orca`—request custom permissions defined by other pre-installed apps signed by the same certificate, hence most likely Facebook apps too. Such permissions include `com.facebook.appmanager.ACCESS` or `com.facebook.receiver.permission.ACCESS`, which are not publicly documented. It is possible that these permissions are potentially related to partnerships and data-sharing practices between Facebook and OEMs as revealed in 2018 by the New York Times [81].

TABLE 3: Number of custom permissions *definitions* that do not follow the naming convention. Note that an application defining multiple custom permissions will be counted multiple times in this table.

Origin	# of definitions	# of bad definitions	Percentage
Google Play	63,193	7,087	11%
Tencent	9,902	1,629	17%
APKMonk	3,060	298	10%
Xiaomi Mi	5,898	1,219	21%
Baidu	4,703	612	13%
APK Mirror	19,543	1,654	9%
Huawei	3,392	464	14%
Qihoo 360	1,999	297	15%
AndroZoo (other stores)	28,636	9,478	33%
Pre-installed	2,237,585	1,045,815	47%
Total	**2,373,124**	**1,067,421**	**45%**

6 Naming and Declaration Conventions

Google recommends app developers to define custom permissions following a clear naming convention and to add a description of the purpose of the custom permission. We find that this recommendation is not enforced. Figures 8a and 8b show the scale and complexity of the problem for a subset of custom permissions that are requested by at least 2,000 apps in our dataset. Using the attribution methodology described in Section 4.1, we cluster this subset of popular custom permissions into 67 second-level domains (SLDs). When analyzing all custom permissions in our dataset, we find a total of 11,209 SLDs groups, the majority of which (65%) only contain one custom permission, and 94% five or less. Without proper and verifiable naming conventions, nor a clear description of the services and data protected by custom permissions, users cannot take informed decisions when granting custom permissions to apps. In fact, a malicious app developer could easily confuse users by (intentionally) impersonating a well-known prefix, such as `com.google` or `com.samsung`. In this section, we empirically measure whether app developers exposing permissions follow recommended practices.

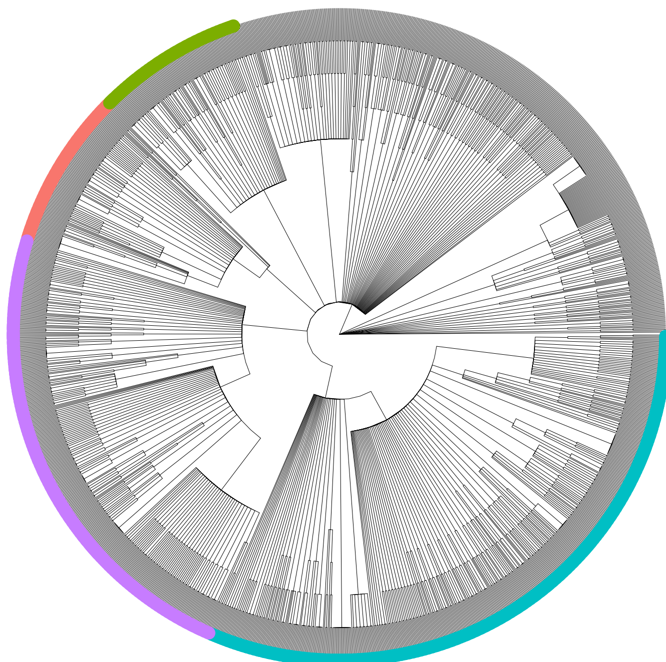
6.1 Naming convention violations

We find naming convention violations to be widespread. Table 3 lists the percentage of definitions that fail to adhere to the naming convention, broken down per origin. The percentage of permission declarations that fail to adhere to the naming convention varies from 8% to 33% on public app stores. For pre-installed apps, almost half (47%) of custom permission declarations break the naming convention.

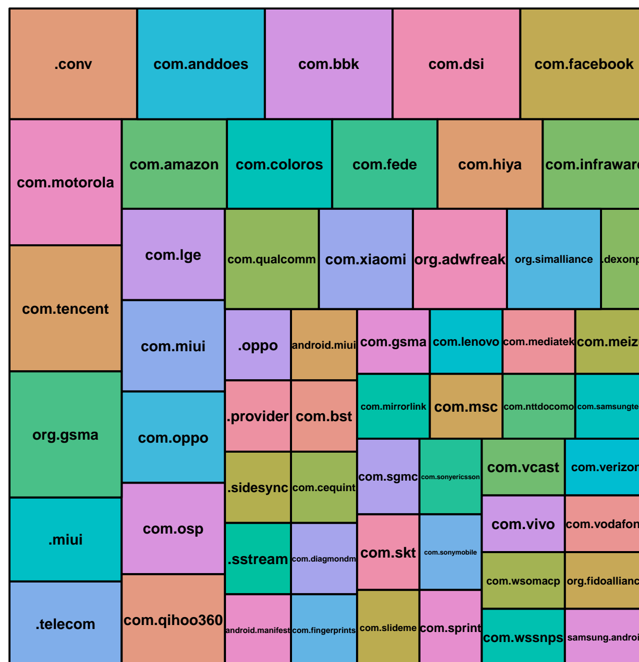
An example of such a violation is the `com.qualcomm.permission.QCOM_AUDIO` permission, defined by the

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981



(a) Phylogenetic tree. The colors represent the most common SLDs: 🔴 com.google, 🟢 com.huawei, 🟣 com.sec, 🔵 com.samsung Note that the com.sec prefix might in fact be related to Samsung's Knox API [35]



(b) Treemap. For readability, we do not include the top 10 most common SLDs. The excluded prefixes seems to be associated with Samsung (com.samsung, com.sec, .sec), Google (com.google), Huawei (com.huawei, .huawei), HTC (com.htc), and other entities which we could not identify (android.permission, com.android, org.adw)

Fig. 8: Treemap and phylogenetic tree of custom permissions requested by at least 2,000 apps each, grouped by their second level domain.

com.verizon.obdm_permissions app. Not only are the SLDs of the package name (qualcomm.com, a chipset manufacturer) and of the custom permission (verizon.com, a network operator) different, but the Subject field of the signing certificate of the app mention a third entity, Google. In that case, it is impossible to attribute with certainty the custom permission to any of these entities.

Some violations are due to developers choosing to use the same prefix as AOSP permissions, which can also confuse the end user into granting a permission, thinking it was created by the operating system, such as android.permission.DOWNLOAD_WITHOUT_NOTIFICATION, or android.permission.RECORD_VIDEO. In total, we find 722 custom permissions that use the android.permission prefix. This high number of permissions using AOSP prefixes is surprising as OEMs are explicitly forbidden from adding permissions to the android.* namespace as part of their customization of the OS [5]. Yet, we find that 87% of the apps defining at least one of the 722 custom permissions that we identified are pre-installed applications, which could be a breach of the CDD. This issue is still present in recent versions of Android: we find that 226 of these permissions (31% of the total) are defined by apps pre-installed on devices running Android 11 or 12. Anecdotally, we observe instances of applications requesting custom permissions with names that are similar to those of well-known AOSP permissions, but with typos. We find, for instance, custom permissions that include the string andorid instead of android, CORSE_LOCATION instead of COARSE_LOCATION, or RUN_TIME instead of RUNTIME.

We also find evidence suggesting that some naming violations might be due to embedded third-party SDKs or components integrated in the app: if an app embeds an SDK that defines a custom permission, that permission will be in the manifest of the host app (as explained in Section 2), and most likely result in a violation of the naming convention (unless both the app and the SDK share the same package name). For instance, the app com.iugome.lilknights (an RPG game available on Google Play) defines the permission com.facebook.orca.provider.ACCESS, which seems to be associated with the Facebook Messenger app. Another more complex example is the com.verizon.permission.ACCESS_REMOTE_SIMLOCK permission, defined by the com.mediatek.op12.phone app. Not only are the SLDs of the package name (mediatek.com, a chipset manufacturer) and of the custom permission (verizon.com, a network operator) different, but the signing certificate of the app mentions a third entity: TCL, a phone manufacturer. Unfortunately, the lack of developers' compliance and third-party control by app markets defeats any automatic effort to perform accurate attribution of custom permissions to the responsible party.

6.2 Documentation for custom permissions, or lack thereof

One option to better understand custom permissions would be to look at their descriptions on the Android Manifest

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

TABLE 4: Percentage of custom permissions definitions (grouped by their SLD or not) without description

Origin	% of definitions without description	% of SLDs without description
Google Play	82%	75%
Tencent	94%	91%
APKMonk	76%	66%
Xiaomi Mi	91%	88%
Baidu	98%	97%
APK Mirror	74%	48%
Huawei	97%	94%
Qihoo 360	96%	93%
AndroZoo (other stores)	67%	60%
Pre-installed	70%	45%
All	75%	47%

file. While documenting custom permissions is a practice recommended by Google [47], it is not mandatory for developers and we find that in 75% of the cases this field is just empty. Table 4 shows the percentage of custom permissions definitions without description broken down by the origin of the apps. We also give the percentage of custom permissions without description when grouped by their prefix SLD. As it can be observed, applications very often lack custom permissions' description when regardless of their origin market.

We also find that when developers provide a custom permission description, it is often vague and does not describe accurately what their actual purpose is (e.g., "Quick connect" or "Dolby Tuning permission description"). In some cases, the suffix of a permission can render useful for inferring their purpose. We find custom permissions that use the exact same suffix as official AOSP permissions, such as `com.oppo.permission.safe.CAMERA` or `thinkyeah.permission.READ_SMS`. In total, we find 142 unique custom permissions with a normal protection level that use the same suffix as a `dangerous` AOSP permission, and 1,334 with a `signature` or `signatureOrSystem` suffix. It is unclear to us why these developers might try to replicate AOSP permissions, and this might suggest that they could provide covert access to AOSP-protected system resources and data. Nonetheless, such a string-based analysis is not conclusive in itself, and requires further code-level investigation.

Finally, we find that online documentation explaining which company is behind a given permission and what is the functionality or data protected is very scarce. In fact, we manually looked for public documentation for the permissions in our dataset using online search engines and do not find publicly available documentation for most of them (94%). This is a highly manual and time-consuming task, and thus we could not realistically manually search for 257,710 permissions. Instead, we rank the permissions by their prevalence and focus our manual efforts on those that are most highly used. For the lesser known permissions, we implement an automatic crawler that relies on the DuckDuckGo API to search for documentation relevant to the permission. Furthermore, we also crawl StackOverflow forums to find discussions revolving around the permission. Even when combining automatic and manual analysis of

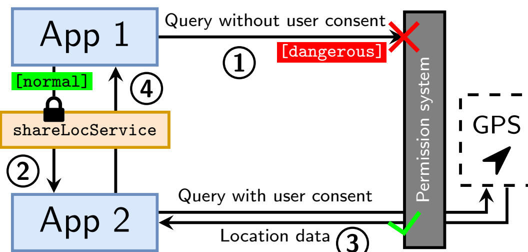


Fig. 9: Scenario where an attacker bypasses the permission model using a service protected by a custom permission. The circled numbers indicate the order of each step.

different resources, we are barely ever able to find any information relevant to a given permissions functionality. This suggests that inferring the purpose of a custom permission requires analyzing the code of the app.

7 Detecting Leaky Custom Permissions

The main goal of the Android permission system is to protect sensitive system APIs from unwanted access without explicit user consent. However, custom permissions also make the Android permission model vulnerable to an elevation of privilege attack, as highlighted by Tuncay et al. [87] and Bagheri et al. [63], [64]. In this scenario, we hypothesize that an app can obtain access to sensitive data, or to perform an action that is protected by an AOSP permission, and then make it available to other apps via a custom permission that has a lower protection level than the original AOSP permission.

Figure 9 illustrates this situation. App_1 first tries to get the user's location through the official API but either lacks the necessary AOSP permission or the user rejects the request, so it is denied. Then, App_1 sends an `Intent` [57] to the `shareLocService` service exposed by App_2. This component is protected by a custom permission that App_1 holds. App_2 also holds the AOSP location permission, so it is able to successfully obtain the user's location. App_2 then sends back the location to App_1 as a response to its `Intent`.

In this particular scenario, App_1 and App_2 do not necessarily need to cooperate. The result is identical if App_2 fails to correctly protect its service, e.g., by giving access to it with a permission that has a `normal` protection level. This creates a vulnerability that an attacker could exploit simply by sending an `Intent` to the service to retrieve the location. In the attack above, the only user interaction that will occur would be at step three, where the OS will display a popup window to ask the user if they wish to allow App_2 to access the location. If the user had already granted such a permission to App_2, then the attack will play out without any user interaction.

7.1 Tooling

Android's custom permissions are asynchronous software artifacts that are difficult to monitor, model, and study. While there is a vast arsenal of highly useful static and dynamic analysis tools to study many harmful and privacy-intrusive behaviors on Android, none of them are fit to effectively infer the purpose of custom permissions and to

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

determine whether they expose sensitive data or system resources. For example, Flowdroid [60] allows tracking data flows within a given component, but it is unable to handle neither inter-component nor inter-app communication—both of which are essential in the analysis of custom permissions. Similar limitations are present in Amandroid (since renamed Argus-SAF) [89] which is able to detect inter-component leaks but it does not detect information leaks between apps through components protected by a custom permission. Finally, PScout [61] analyzes permissions by mapping them to AOSP APIs, and it is not intended for understanding what these permissions are protecting or for determining the purpose of a custom permission. Furthermore, typical analysis challenges such as software obfuscation, dynamic code loading, or deodexing of compiled pre-installed software further complicate the analysis of custom permissions.

To overcome these technical limitations and challenges, we create `permissionTracer` and `permissionTainter`, two complementary tools tailored to the analysis of custom permissions:

Tool 1. `permissionTracer`. We create `permissionTracer`, a triage tool to extract information about the data type or features protected by custom permissions. Given an application defining custom permissions, the tool analyzes all components protected by such permissions and reports: (i) the data types of return values and method prototypes that an app can access when interacting with said component; and (ii) the list of APIs protected by AOSP permissions accessed within the component's methods. The ability to extract this knowledge allows determining whether components protected by custom permissions could potentially allow access—by mistake or by design—to restricted data to an app that does not hold the required AOSP permission and which ones might require manual verification. The way `permissionTracer` analyzes a protected component depends on its type:

- For activities and broadcast receivers, it looks for the `setResult` method and extracts its return data type.
- For content providers (which work as a database for other applications), `permissionTracer` obtains the type of the `getType` method.
- For services where no data is returned, it extracts and parses the method prototypes (i.e., method name, return type, and parameter types) from all the interfaces that are returned by the `onBind` method. The type of data (e.g., `String` or Android objects such as `Android.location.Location`) allows understanding the kind of information (e.g., contacts or location) it might expose.

`permissionTracer` follows a tree search of all method calls and parses the Smali code of each method looking for API calls. This process involves multiple steps. First, `permissionTracer` extracts and classifies all methods as either *external*, i.e., not defined by the app being analyzed like AOSP calls, or *internal*. For the external calls, `permissionTracer` looks at whether an AOSP permission is needed to invoke the method using our permission mappings.[3] For internal methods, it adds them to a stack and traverses them recursively once the current method has been analyzed. We limit the stack size to an arbitrary limit of 7 method calls. To that end, we modify Androguard [2] to load our AOSP permission mappings, and to obtain the list of permission protected APIs accessed in a given class. We evaluate `permissionTracer` by manually inspecting the Android components protected by custom permissions across 400 APKs. From those, we manually extract the objects and value types that the components return, and compare this to the output of `permissionTracer` in the dataset. We do not find any false positive or false negative in the output of our tool. We make our modifications publicly available along with `permissionTracer`'s code and AOSP permission mappings.

Tool 2. `permissionTainter`. `permissionTracer` cannot discover potential leaks of data protected by AOSP permissions. To aid in this task, we build `permissionTainter`, a static taint analyzer developed to study custom permissions on top of our modified version of Androguard. `permissionTainter` starts by looking for intent filters that are registered by the application that are not already defined in the app's manifest. Then, it parses the DEX code to look for intents and handlers, and tries to associate them with their target. For intents, the target can be explicitly set by the app, or implicit in the case of broadcasted intents. In the latter case, we use the list of intent filters to determine the classes that would receive such an intent.

After this step, `permissionTainter` enriches the analysis object created by Androguard (which contains, among other elements, all the classes, methods and the cross-references between them) to add extra cross-references to account for asynchronous communications, such as intents. Essentially, `permissionTainter` creates a graph representing the whole DEX code where vertices are methods and edges are methods calls, which now include asynchronous communications as well.

Finally, `permissionTainter` relies on the default sources and sinks used by Flowdroid [23] along with the modifications shown in Table 7 of the appendix. It also considers any AOSP API protected by an AOSP permission as a source. `permissionTainter` first locates all calls to sink methods and, for each occurrence, builds a call graph rooted at that method. It then looks for any call to a source method in that call graph and extracts all paths from the sources to the sinks. A path in the call graph indicates that the value returned by the source method could make its way to the sink. `permissionTainter` then follows each path in the call graph and creates the corresponding control flow graph (CFG). Again, `permissionTainter` looks for

3. To extract the list of protected AOSP APIs, we update Axplorer's mappings [62] with (1) mappings provided by Android Studio IDE [8]; This includes lint scripts to warn developers if they use certain API calls without requesting the associated permission. and (2) knowledge extracted from the AOSP source code to see the prototypes of methods that use the `@RequiresPermission` annotation [42], which indicate the permission(s) that need to be granted to an app in order to invoke a given AOSP method. To the best of our knowledge, we are the first to follow this easy-to-update approach to obtain a more complete and fresh mapping of API calls to AOSP permissions.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

all paths from the source to the sink, this time in the CFG, and applies tainting rules to detect potential misuses.

Limitations. Both of our tools suffer from a number of limitations which are common to other static analysis approaches for Android apps. They cannot detect calls to protected APIs that are called by other components loaded dynamically during runtime (e.g., using Java's reflection [58] or JNI APIs). While they can tell if a component uses permission-protected APIs, they cannot guarantee that the component will be actually used in runtime. Moreover, pre-installed applications can use ODEX instead of DEX files, which are stored alongside the APK file. Because of limitations in our data collection strategy, we may miss the ODEX file associated with an APK, which prevent us from doing any code analysis. Lastly, our tools cannot detect if an app manually implements access control mechanisms (e.g., by checking the package name of the calling app upon receiving an intent). Such an analysis must therefore be conducted manually, after detection of a potential case of abuse.

7.2 Analysis results

We run both tools on our dataset of 96,748 unique applications exposing custom permissions to protect 214,943 components.[4] Using `permissionTracer`, we find that 24,648 of those components (11%) access at least one API protected by an AOSP permission, and 16% of those components access at least one API protected by an AOSP permission with a `dangerous` protection level. This tool allows us to identify the following behaviors:

Sensitive components

We find that 1,209 components (over 2,192 apps) use a custom permission with a `normal` protection level. These components are essentially unprotected, as the `normal` protection level allows any app on the device to request and be granted the permission. 55% of these apps are pre-installed. For example:

- 950 of those components access APIs protected by the `READ_PHONE_STATE` permission, which grants access to non resettable device identifiers such as the IMEI until Android 10, which can be used for user tracking [16].
- 497 components access location data protected by the `ACCESS_COARSE_LOCATION` permission, while 422 access `ACCESS_FINE_LOCATION`.
- We find 134 components accessing APIs protected by `READ_PRIVILEGED_PHONE_STATE`, which also gives access to unique identifiers, and 58 components accessing APIs protected by `WRITE_SECURE_SETTINGS` which allows for the modification of the system preferences of the device.

Such findings do not necessarily indicate a malicious intent from the developer, but insecure development practices that could be exploited by malicious actors to access AOSP-protected data without user awareness. This is particularly concerning with `normal` custom permissions, which are granted automatically at install time. An example of such a permission is `melons.dialer.permission.CALL_LOG`, defined by a dialer app that was published on Google Play. This permission has a `normal` protection level and protects a content provider that allows other apps to read and delete entries from the call log. The application implements access control simply by checking the package name of the caller app, and only allows queries from package names in a hard-coded list of messenger apps, including some from the same developer. Thus, an attacker just needs to use one of these packages names for their app and then query the dialer app to read or delete call log entries without requesting the AOSP permission. We tested and verified this vulnerability dynamically with a proof-of-concept app.

We also study in detail the return types of the 3,780 methods that `permissionTracer` detected. Unsurprisingly, we find that most methods return void, boolean, or integer values (36%, 29% and 16% of the cases, respectively). However, the method returns Android objects in 123 cases. For instance, the `mobi.maptrek.lite` app defines the `mobi.maptrek.lite.permission.RECEIVE_LOCATION` permission (`normal` protection level) to protect a service that defines a `getLocation()` method, which returns a `Landroid/Location/Location` object. Further analysis of the app code shows that the service makes the user location available to any colluding application that requires the custom permission. The app defining this permission is an offline map app, intended to be used during outdoor activities when the user has no Internet connection. The app is available on Google Play Store and has been downloaded over 10k times. We verified this attack with a proof-of-concept app, showing that any app can access the user location without requesting the official AOSP permission and without the need to interact with the developers of the other app. In 19% of the cases, the methods return a custom object defined by the app itself.

PII leaks

`permissionTainter` detects 5 potential PII leaks in pre-installed applications. All these apps implement a similar pattern: upon receiving an intent with a specific action (which can be discovered by simply analyzing the source code of the protected component), an attacker can make the component broadcast an intent which contains the Wi-Fi and Bluetooth MAC addresses as extras. We find these apps even in recent Samsung, Asus and LGE devices running Android version 11. We have not found similar behaviors in apps published in app stores. Any colluding app that has the correct intent filter (which can also be simply discovered by analyzing the component's source code) can then receive that intent and get access to the MAC addresses. The MAC addresses can then be used to uniquely identify a user, or can be used to infer their location [34].

Placeholder permissions

We identify 212,277 applications defining custom permissions that are potentially unused, i.e., the permissions is defined but it is never used in the manifest to protect any of the app's components. We name those as *"placeholder permissions."* The reasons why they are defined remain unknown

4. Note that given the scale of our dataset, we only analyse the latest version of each package and, in the case of pre-loaded apps, we define as unique apps those unique combinations of package names and signing certificates.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

TABLE 5: Number of apps defining placeholder permissions and apps dynamically enforcing custom permissions broken down by dataset of origin.

Origin	# placeholder applications	# calling `check*`	# calling `enforce*`	# calling any
Pre-installed	189,177	45,889	5,771	51,793
Public apps	23,143	149	8	149
Total	**212,277**	**5,779**	**46,038**	**51,942**

to us but it might be the result of poor development practices, such as including code obtained from online forums or legacy code from older versions of the app. Yet, it is possible that such apps do not rely on the system's package manager to enforce their permission and chose to do so internally using either `checkPermission`, `enforcePermission`, or one of their variants [56].

To detect such cases, we analyze the binaries of these apps to look for calls to these methods. We find stark differences between pre-installed apps, where 51,793 of the apps call one of the methods, and publicly-available apps, where only 149 of the apps do so. Overall, only 51,942 of the apps seem to do dynamic enforcement of custom permissions. Table 5 shows the number of apps for which we detected at least one call to `checkPermission`, `enforcePermission` or one of their variants [56] in the DEX or ODEX code of apps that are defined but do not protect any component. We grouped together all apps collected from public app stores or from AndroZoo under the "Public apps" category.

To gain a better understanding of why so many app developers define custom permissions but do not protect any component with it (nor enforce them dynamically), we contacted 529 developers using the contact email address listed in the public profile of their apps. We discuss the ethical considerations and IRB approval in §4. Our survey received 53 responses. Surprisingly, 28% of the developers that responded to us either did not know that their app defined a custom permission or they did not know why it was there. In 17% of the cases, an SDK used by the developer added the permission. In 9% of the cases, the permission was associated with an old feature that had been already removed.

Although the scale of our survey is small, it provides some intriguing perspectives on the reasons behind the widespread usage of custom permissions. The responses suggest a poor understanding of the (custom) permission system by some developers, which could negatively impact users by inadvertently exposing sensitive data or resources.

8 Discussion

Through the course of this research we have uncovered several problems inherent to custom permissions. As they are, custom permissions open various avenues for abuse, an issue which is compounded by a severe lack of transparency in the app ecosystem of Android. This stems chiefly from a lack of enforcement of software development and platform policies that promote transparency and best development practices. As a result, and as we demonstrated, finding abuse and errors in custom permissions, as well as attributing behaviors, is a herculean task, resulting in a lack of accountability across the ecosystem. In this section, we discuss our findings and propose workable solutions based on our observations.

Google, both as the platform operator and the main driving force behind AOSP, is in a privileged position to mitigate the issues we reported about the use and abuse of custom permissions. Google has already fixed previously-discovered issues, such as the permission re-delegation and confused deputy attacks described by Tuncay et al. [87] but the technical challenges imposed by custom permissions have impeded the discovery of insecure implementations as the ones described in this paper. We next discuss potential strategies to address the issues reported in this paper.

8.1 Privilege escalation

Fixing privilege escalation issues arising from custom permissions is complex, as it exploits a functionality inherent to the Android permission system. One approach to tackle it would be to determine the AOSP permissions being used in the protected component to perform a risk assessment using a tool such as `permissionTracer`. Note that a dangerous permission might, nonetheless, be used within a component without exposing the data protected by it. We believe that the ability to automatically prevent potential attacks justifies instances in which the platform enforces a higher permission level (e.g., dangerous) for a custom permission than the originally necessary (e.g., normal). This enforcement can be done automatically by analyzing the app's code and it could be introduced as part of the analysis processes implemented in Google Play Protect [59], the built-in security mechanism present on Android devices and in the Google Play Store.

8.2 Transparency and user control

Requiring app developers to include a better description of the purpose and the potential risks associated with their defined custom permissions is an important and much-needed first step to improve transparency, promoting user awareness, and empowering user control. We understand that developers can still be obscure or deceitful in describing the purpose of a permission. To make this more effective, we suggest extending the description field with a mandatory risk self-assessment done by the developer. Such assessment might consist of a few key questions with a set of predefined answers regarding the data and features accessed or shared by the permission. Software distribution channels can verify and enforce permission description sanity, at least at a basic level. Furthermore, this could be a way to ensure that developers do not define custom permissions that are unnecessary, reinforcing the practices already implemented by Google to encourage developers to minimize the access to sensitive permissions via permission nudges [82].

Another step in the right direction would be to inform users about the custom permissions requested and defined by an app. Right now, a custom permission is only shown to the user when requested, if the developer itself decides to give it a dangerous protection level. The risk self-assessment discussed above should be the basis to convey the information effectively. The replies to the set of questions could be leveraged to automatically decide the protection level of the custom permission, instead of leaving this decision to the

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

developer. Finally, the platform should offer users a mechanism to revoke previously granted custom permissions, both individually for a particular app or globally within the system through a blocklist.

8.3 Accountability

The attribution problem in Android extends beyond custom permissions as it is rooted in the absence of a reliable way of tracing an app back to its developer. One potential solution to this problem would be for Google to require app developers to take ownership of their apps through a centralized certificate solution. This, in turn, allows users to know the true developer of the apps, as well as the entity that exposes the custom permission to other apps (which itself could be an embedded third-party component). Additionally, custom permissions should add a *definer* tag to their definitions so that a user would always know who is the actor behind a given custom permission as in the case of permissions defined by third-party components embedded in the app.

9 Conclusions

In this paper, we presented a holistic view of the prevalence of custom permissions in the Android ecosystem and their inherent transparency, security and privacy problems. Our findings suggest that, despite this being a widely used feature in both pre-installed and publicly available apps, custom permissions lack transparency, accountability, and it is the source of potential security and privacy harm for end users. We hope that our work will bring more focus to the issues surrounding Android's custom permissions ecosystem. In an effort to foster more research efforts in this area, we make available our dataset of custom permissions [21], [20], as well as the source code of our tools, `permissionTracer` [38] and `permissionTainter` [39], to the research community, platform operators, and regulators.

Acknowledgments

This research has been partially funded by the Spanish Government grant ODIO (PID2019-111429RB-C21 and PID2019-111429RBC22); the Region of Madrid, co-financed by European Structural Funds ESF and FEDER Funds, grant CYNAMON-CM (P2018/TCS-4566); Google and Consumer Reports; and by the EU H2020 grant TRUST aWARE (101021377). The opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect those of any of the funders.

References

[1] Amazon Push Notification Service. https://developer.amazon.com/docs/adm/integrate-your-app.html. [Online; accessed 03-Feb-2021].
[2] Androguard. https://github.com/androguard/androguard/. [Online; accessed 19-March-2019].
[3] Android Certified Partners — brands. https://www.android.com/certified/partners/. [Online; accessed 7-July-2020].
[4] Android Certified Partners — ODMs. https://www.android.com/certified/partners/#tab-panel-odms. [Online; accessed 7-July-2020].
[5] Android Comptability Document — Permissions. https://source.android.com/compatibility/android-cdd#9_1_permissions. [Online; accessed 7-July-2020].
[6] Android Developers. https://developer.android.com/guide/topics/manifest/permission-element.html. [Online; accessed 29-May-2019].
[7] Android Developers - Define a Custom App Permission. https://developer.android.com/guide/topics/permissions/defining. [Online; accessed 25-Jul-2019].
[8] Android Studio code annotations. https://android.googlesource.com/platform/tools/adt/idea/+/refs/heads/mirror-goog-studio-master-dev/android/annotations/android/. [Online; accessed 23-March-2020].
[9] Android Version Distribution statistics. https://www.xda-developers.com/android-version-distribution-statistics-android-studio/. [Online; accessed 31-May-2020].
[10] Androidmanfiest.xml. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/res/AndroidManifest.xml. [Online; accessed 29-July-2019].
[11] android:sharedUserId. https://developer.android.com/guide/topics/manifest/manifest-element#uid. [Online; accessed 15-Mar-2021].
[12] APK Mirror App Store. https://www.apkmirror.com/. [Online; accessed 28-May-2020].
[13] APK Monk App Store. https://www.apkmonk.com/. [Online; accessed 28-May-2020].
[14] Baidu App Store. https://shouji.baidu.com/. [Online; accessed 28-May-2020].
[15] Baidu Push Notification Service. http://push.baidu.com/doc/android/api. [Online; accessed 03-Feb-2021].
[16] Best practices for unique identifiers. https://developer.android.com/training/articles/user-data-ids. [Online; accessed 1-Apr-2021].
[17] Cars — Android Developers. https://developer.android.com/reference/android/car/Car.
[18] Commit a90c8de: Add new "preinstalled" permission flag. https://android.googlesource.com/platform/frameworks/base/+/a90c8def2c6762bc6e5396b78c43e65e4b05079d. [Online; accessed 11-July-2019].
[19] Dataset for Sources and Sinks. https://github.com/Android-Observatory/PermissionTainter/blob/master/SourcesAndSinks_custom_perms.txt.
[20] Dataset of defined custom permissions. https://androidobservatory.com/files/defined_perms_all_release.json.xz.
[21] Dataset of requested custom permissions. https://androidobservatory.com/files/requested_perms_all_release.json.xz.
[22] Firmware Scanner. https://play.google.com/store/apps/details?id=org.imdea.networks.iag.preinstalleduploader. [Online; accessed 06-March-2019].
[23] FlowDroid's Sources and Sinks list. https://github.com/secure-software-engineering/FlowDroid/blob/develop/soot-infoflow-android/SourcesAndSinks.txt.
[24] Google Maps Receive Permission. https://stackoverflow.com/questions/14832911/android-map-v2-why-maps-receive-permission. [Online; accessed 03-Feb-2021].
[25] Google Play App Store. https://play.google.com/store/apps/. [Online; accessed 28-May-2020].
[26] Google Push Notification Service. https://web.archive.org/web/20121004073640/https://developers.google.com/android/c2dm/. [Online; accessed 03-Feb-2021].
[27] Huawei App Store. https://appgallery1.huawei.com/#/Featured. [Online; accessed 28-May-2020].
[28] Huawei Push Notification Service. https://stackoverflow.com/questions/57860791/how-to-access-payload-of-hms-push-notifications. [Online; accessed 03-Feb-2021].
[29] Huawei's Android App Store Launches Internationally. https://www.androidheadlines.com/2018/04/huaweis-android-app-store-launches-internationally.html. [Online; accessed 16-June-2020].
[30] Integrate Amazon Device Messaging (ADM). https://developer.amazon.com/docs/video-skills-fire-tv-apps/integrate-adm.html.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

[31] Jiguang Push Notification Service. https://docs.jiguang.cn/en/jpush/client/Android/android_guide/#configuration-and-code-instructions. [Online; accessed 03-Feb-2021].
[32] Market share development per Android phone manufacturer. https://www.appbrain.com/stats/top-manufacturers.
[33] Merge multiple manifest files. https://developer.android.com/studio/build/manifest-merge. [Online; accessed 22-July-2020].
[34] Mobile Advertising Network InMobi Settles FTC Charges It Tracked Hundreds of Millions of Consumers' Locations Without Permission. https://www.ftc.gov/news-events/news/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked-hundreds-millions-consumers.
[35] New permissions names. https://docs.samsungknox.com/dev/knox-sdk/new-permission-names.htm.
[36] permission — Android Developers. https://developer.android.com/guide/topics/manifest/permission-element#desc.
[37] Permissions. https://docs.samsungknox.com/dev/common/license-permissions.htm.
[38] PermissionTainter. https://github.com/Android-Observatory/PermissionTainter.
[39] PermissionTracer. https://github.com/Android-Observatory/PermissionTracer.
[40] Qihoo 360 App Store. http://zhushou.360.cn/. [Online; accessed 28-May-2020].
[41] R.attr. https://developer.android.com/reference/android/R.attr.html#protectionLevel. [Online; accessed 2-July-2019].
[42] RequiresPermission — AndroidX. https://developer.android.com/reference/androidx/annotation/RequiresPermission. [Online; accessed 23-March-2020].
[43] Samsung Knox. https://www.samsungknox.com/en. [Online; accessed 15-Apr-2021].
[44] Tencent App Store. https://android.myapp.com/. [Online; accessed 28-May-2020].
[45] Xiaomi Mi App Store. http://app.mi.com/. [Online; accessed 28-May-2020].
[46] Xiaomi Push Notification Service. https://docs.moengage.com/docs/android-xiaomi-push. [Online; accessed 03-Feb-2021].
[47] Android Developers - Define a Custom App Permission, 2018. https://developer.android.com/guide/topics/permissions/defining.
[48] Android Developers - GoogleSignInApi. https://developers.google.com/android/reference/com/google/android/gms/auth/api/signin/GoogleSignInApi, 2018. [Online; accessed 6-Aug-2020].
[49] Android Developers - Permissions Overview, 2018. https://developer.android.com/guide/topics/permissions/overview.
[50] AndroZoo, 2018. https://androzoo.uni.lu/.
[51] Google Issue Tracker - Why Google play services dependency automatically added com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE permission. https://issuetracker.google.com/issues/78380811#comment22, 2018. [Online; accessed 6-Aug-2020].
[52] Migrate a GCM Client App for Android to Firebase Cloud Messaging. https://developers.google.com/cloud-messaging/android/android-migrate-fcm, 2018.
[53] Android Developers - Play Install Referrer Library. https://developer.android.com/google/play/installreferrer/library, 2019. [Online; accessed 6-Aug-2020].
[54] Android Developers - AIDL to Google Play Billing Library migration guide. https://developer.android.com/google/play/billing/migrate, 2020. [Online; accessed 6-Aug-2020].
[55] Android Developers - Sign your app. https://developer.android.com/studio/publish/app-signing, 2020. [Online; accessed 25-Aug-2020].
[56] Context - Android Developers. https://developer.android.com/reference/android/content/Context.html, 2020.
[57] Intent - Android Developers. https://developer.android.com/reference/android/content/Intent, 2020.
[58] Using Java Reflection. https://www.oracle.com/technical-resources/articles/java/javareflection.html, 2020.
[59] https://developers.google.com/android/play-protect. https://developers.google.com/android/play-protect, 2021. [Online; accessed 15-Apr-2021].
[60] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Proceedings of the ACM Special Interest Group on Programming Languages (SIGPLAN)*, 2014.
[61] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, 2012.
[62] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *Proceedings of the USENIX Security Symposium*, 2016.
[63] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. Detection of design flaws in the android permission protocol through bounded verification. In *Proceedings of the International Symposium on Formal Methods*, 2015.
[64] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. A formal approach for detection of security flaws in the android permission system. *Formal Aspects of Computing*, 2018.
[65] Kenneth Block, Sashank Narain, and Guevara Noubir. An autonomic and permissionless android covert channel. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2017.
[66] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.
[67] Saksham Chitkara, Nishad Gothoskar, Suhas Harish, Jason I Hong, and Yuvraj Agarwal. Does this app really need my location? context-aware privacy management for smartphones. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2017.
[68] Luke Deshotels. Inaudible sound as a covert channel in mobile devices. In {*USENIX*} *Workshop on Offensive Technologies* ({*WOOT*}), 2014.
[69] David Dittrich and Erin Kenneally. The Menlo Report: Ethical principles guiding information and communication technology research. *US Department of Homeland Security*, 2012.
[70] Álvaro Feal, Julien Gamba, Juan Tapiador, Primal Wijesekera, Joel Reardon, Serge Egelman, and Narseo Vallina-Rodriguez. Don't accept candy from strangers: An analysis of third-party mobile sdks. *Data Protection and Privacy: Data Protection and Artificial Intelligence*, 2021.
[71] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, 2011.
[72] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the USENIX conference on Web application development*, 2011.
[73] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2012.
[74] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the USENIX Security Symposium*, 2011.
[75] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An analysis of pre-installed android software. *IEEE Symposium on Security and Privacy (SP)*, 2020.
[76] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*, 2012.
[77] Kaspar Hageman, Álvaro Feal, Julien Gamba, Aniketh Girish, Jakob Bleier, Martina Lindorfer, Juan Tapiador, and Narseo Vallina-Rodriguez. Mixed signals: Analyzing software attribution challenges in the android ecosystem. *IEEE Transactions on Software Engineering*, 2023.
[78] Kristen Kennedy, Eric Gustafson, and Hao Chen. Quantifying the effects of removing permissions from android applications. In *Mobile Security Technologies (MoST)*, 2013.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

[79] Rui Li, Wenrui Diao, Zhou Li, Jianqi Du, and Shanqing Guo. Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings. 2021.

[80] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *Proceedings of the International Conference on Software Engineering*, 2016.

[81] New York Times. Facebook Gave Device Makers Deep Access to Data on Users and Friends. https://www.nytimes.com/interactive/2018/06/03/technology/facebook-device-partners-users-friends-data.html.

[82] Sai Teja Peddinti, Igor Bilogrevic, Nina Taft, Martin Pelikan, Úlfar Erlingsson, Pauline Anthonysamy, and Giles Hogben. Reducing permission requests in mobile apps. In *Proceedings of the Internet Measurement Conference (IMC)*, 2019.

[83] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[84] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions systems. *Proceedings of the USENIX Security Symposium*, 2019.

[85] Alireza Sadeghi, Hamid Bagheri, and Sam Malek. Analysis of android inter-app security vulnerabilities using covert. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 725–728. IEEE, 2015.

[86] James Sellwood and Jason Crampton. Sleeping android: The danger of dormant permissions. In *Proceedings of the ACM workshop on Security and Privacy in Smartphones & Mobile Devices*, 2013.

[87] Güliz Seray Tuncay, Soteris Demetriou, Karan Ganju, and C Gunter. Resolving the predicament of android custom permissions. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[88] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond Google Play: A Large-Scale Comparative Study of Chinese Android App Markets. In *Proceedings of the Internet Measurement Conference (IMC)*, 2018.

[89] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)*, 2018.

[90] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android permissions remystified: A field study on contextual integrity. In *Proceedings of the USENIX Security Symposium*, 2015.

Julien Gamba is a PhD student in the Internet Analytics Group at the IMDEA Networks Institute. His research revolves around user's security and privacy in Android devices. In his work, Julien uses both static and dynamic analysis, as well as other techniques specifically designed to understand the behavior of mobile applications. Recently, Julien was the first author of the first large-scale analysis of the privacy and security risks of pre-installed software on Android devices and their supply chain, which was awarded the Best Practical Paper Award at the 41st IEEE Symposium on Security and Privacy. This study was featured in major newspaper such as The Guardian (UK), the New York Times (USA), CDNet (USA) or El País (Spain). Julien was also awarded the ACM IMC Community Contribution Award in 2018 for his analysis of domain ranking services, and was awarded the NortonLifeLock Research Group Graduate Fellowship, the Google PhD Fellowship in Security and Privacy and Consumer Reports' Digital Lab fellowship.

Álvaro Feal is a PhD student working at IMDEA Networks Institute under Prof. Narseo Vallina-Rodriguez's supervision. His research revolves around analyzing privacy threats in the mobile and web ecosystem using static and dynamic analysis techniques as well as network measurements. He has published his research in different venues such as the IEEE Symposium on Security and Privacy, USENIX Security, ACM IMC, PETS Symposium, IEEE ConPro, and CPDP.

Eduardo Blazquez is a PhD student at the Carlos III University of Madrid. His research focuses on Android security and privacy issues, analyzing the Android ecosystem using static and dynamic analysis techniques. Recently, he was the first author of the first analysis of Firmware-over-the-Air applications on Android devices, which was published at the 42nd IEEE Symposium on Security and Privacy.

Vinuri Bandara is a first year PhD student at the IMDEA Networks Institute, supervised by Dr. Narseo Vallina-Rodriguez. Her current research focuses on privacy and security analysis of the android ecosystem along with a focus on privacy policies and regulations. Her research has been published at the IEEE International Working Conference on Source Code Analysis and Manipulation and ACM Conference on Computer and Communications Security.

Abbas Razaghpanah is a Senior Data Scientist at ThousandEyes/Cisco, and a Research Scientist at the International Computer Science Institute (ICSI) at University of California, Berkeley. The crux of his work is the application of network measurements in various areas of networking and security research. His work in the area of mobile privacy and security has been awarded the Distinguished Paper Award at ACM IMC 2018, Best Practical Paper Award at the 41st IEEE Symposium on Security and Privacy, the CNIL-INRIA 2019 award for privacy protection, the 2020 Caspar Bowden Privacy Enhancing Technology Award, and the 2019 AEPD Emilio Aced Prize for Privacy Research. His work on mobile app privacy has received international media attention from The Washington Post, CNET, The Verge, The Guardian, and others.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

Juan Tapiador is Professor of Computer Science at Universidad Carlos III de Madrid, Spain, where he leads the Computer Security Lab. Prior to joining His research interests include binary analysis, systems security, privacy, surveillance, and cybercrime. He has served in the technical committee of conferences such as USENIX Security, ACSAC, DIMVA, ESORICS and AsiaCCS. He has been the recipient of the UC3M Early Career Award for Excellence in Research (2013), the Best Practical Paper Award at the 41st IEEE Symposium on Security and Privacy (Oakland), the CNIL-Inria 2019 Privacy Protection Prize, and th e 2019 AEPD Emilio Aced Prize for Privacy Research. His work has been covered by international media, including The Times, Wired, Le Figaro, ZDNet, and The Register.

Narseo Vallina-Rodriguez is an Associate Research Professor at IMDEA Networks and a co-founder of AppCensus Inc. Narseo obtained his Ph.D. at the University of Cambridge and his research interests fall in the broad areas of network measurements, privacy, and mobile security. His research efforts have been awarded with best paper awards at the 2020 IEEE Symposium on Security and Privacy (S&P), USENIX Security'19, ACM IMC'18, ACM HotMiddlebox'15, and ACM CoNEXT'14 and Narseo has received prestigious industry grants and awards such as a Google Faculty Research Fellowship, a DataTransparencyLab Grant, and a Qualcomm Innovation Fellowship. His research in the mobile security and privacy domain has been covered by international media outlets like The Washington Post, The New York Times, or The Guardian and it has influenced policy changes and security improvements in the Android platform. Narseo's work has received in multiple ocassions the recognition of EU Data Protection Agencies with the AEPD Emilio Aced Award (2019, 2020, and 2021) and the CNIL-INRIA Privacy Protection Award (2019 and 2021). He is also the recipient of the IETF/IRTF Applied Networking Research Award in 2016 and the Caspar Bowden Award in 2020. In 2020, he was awarded a Ramon y Cajal Fellowship by the Spanish Ministry of Science.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3288981

MOST REQUESTED CUSTOM PERMISSIONS

TABLE 6: Top 20 most often requested custom permissions in our dataset, in order. We infer the creator of those permissions using the `Subject` field of the signing certificate of the APKs

Permission name	Creator
com.samsung.android.providers.context.permission.WRITE_USE_APP_FEATURE_SURVEY	Samsung
com.wssnps.permission.COM_WSSNPS	Samsung
com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE	Google Firebase/Android
com.sec.android.diagmonagent.permission.DIAGMON	Samsung
com.sec.android.settings.permission.SOFT_RESET	Samsung
com.sec.android.diagmonagent.permission.PROVIDER	Samsung
com.samsung.android.permission.SSRM_NOTIFICATION_PERMISSION	Samsung
com.sec.phone.permission.SEC_FACTORY_PHONE	Samsung
com.google.android.providers.gsf.permission.READ_GSERVICES	Google Mobile Services/Android
com.samsung.android.bixby.agent.permission.APP_SERVICE	Samsung
com.google.android.gms.auth.api.signin.permission.REVOCATION_NOTIFICATION	Google Mobile Services/Android
com.android.vending.BILLING	Google Mobile Services/Android
com.sec.android.app.twdvfs.DVFS_BOOSTER_PERMISSION	Samsung
com.sec.imsservice.PERMISSION	Samsung
com.sec.imsservice.READ_IMS_PERMISSION	Samsung
com.sec.android.provider.logsprovider.permission.READ_LOGS	Samsung
com.sec.android.provider.badge.permission.READ	Samsung
com.samsung.cmh.data.READ	Samsung
com.sec.enterprise.knox.MDM_CONTENT_PROVIDER	Samsung
com.samsung.android.launcher.permission.READ_SETTINGS	Samsung

ADDED SOURCES AND SINKS USED FOR THE ANALYSIS

TABLE 7: We base the sources and sinks monitored through `permissionTainter` [19] on the default list provided by FlowDroid. We have incorporated the following additions to the default list.

API method signature	Source/Sink
java.lang.Runtime: java.lang.Process exec(java.lang.String)	Source
java.net.HttpURLConnection: void connect()	Sink
java.net.HttpURLConnection: java.io.OutputStream getOutputStream()	Sink
javax.net.ssl.HttpsURLConnection: void connect()	Sink
javax.net.ssl.HttpsURLConnection: java.io.OutputStream getOutputStream()	Sink

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/