Chapter 13
This is Just Metadata: From No Communication Content to User Profiling, Surveillance and Exploitation

Constantinos Kapetanios, Theodoros Polyzos, Efthimios Alepis, and Constantinos Patsakis

Abstract Mobile devices have become an indispensable part of our daily lives. Practically, most of our everyday communication is performed through mobile devices which host third party apps and provide for various means of interaction with diverse levels of security. Android is by far the most widely used mobile operating system, with a user base in the scale of billions. However, while Android Open Source Project (AOSP) is paving the way for all manufacturers, Android market is so fragmented that those who are using the latest version are only a small minority. Moreover, Android comes in several flavours as manufacturers tailor it to their needs. However, this tailoring often prevents users from getting the latest updates. In fact, as we show, manufacturers may not follow the security and privacy guidelines of AOSP, exposing their users to unexpected threats. In this work we study a yet unpatched vulnerability by most major manufacturers, and partially fixed in AOSP, which allows for an adversary to extract important information from the victim's device. To this end, we showcase that unprivileged apps, without actually using any permissions, can harvest a considerable amount of valuable user information. This is achieved by monitoring and exploiting the file and folder metadata of the most well-known messaging apps in Android, which have been hitherto considered secure, deriving thereby usage statistics in order to elicit user profiles, social connections, credentials or other sensitive information.

Keywords Android · Access control · Privacy · Smartphones · Metadata

---

C. Kapetanios · E. Alepis · C. Patsakis (✉)
Department of Informatics, University of Piraeus, Piraeus, Greece
e-mail: kpatsak@unipi.gr

T. Polyzos
Department of Informatics, University of Athens, Athens, Greece

G. A. Tsihrintzis and M. Virvou (eds.), *Advances in Core Computer Science-Based Technologies*, Learning and Analytics in Intelligent Systems 14,
https://doi.org/10.1007/978-3-030-41196-1_13

13.1 Introduction

Back in 2013 when Edward Snowden started unravelling the story behind many secret surveillance projects such as the PRISM,[1] the agencies defended themselves by responding that they were just collecting metadata and not actual content. Therefore, many government agents were arguing that these surveillance actions could not be considered as privacy invasive. For instance, Dianne Feinstein while she was defending NSA phone records program declared [21]:

> As you know, this is just metadata. There is no content involved. In other words, no content of a communication. That can only be, these records, I'm not talking about content, the records can only be accessed under heightened standards.

Consequently, other revelations that followed were far more indicative of the intrusive methods employed in these projects; therefore, the "battle" for the "metadata" argument was soon forgotten. Nevertheless, in this paper we argue that there is far more knowledge in such metadata information than one can even anticipate. Unfortunately, users do not have to protect themselves only from the prying eyes of government surveillance projects, since software companies have already shown similar rogue behaviours under the pretext of user profiling designed towards offering better user experience and personalised recommendations. In fact, the latest methods used for user profiling are so privacy invasive that cannot be considered lightheartedly benign. In the most sinister scenario, usage metadata can be exploited by rogue applications to trick users into providing sensitive information like credentials.

13.1.1 Motivation

Android, having long ago surpassed the boundary of one billion users, is currently by far the most widely used mobile platform [26]. However, Android is fragmented in several aspects. Firstly, while there are many versions (API levels) of Android, notably only a handful of users have access to its latest version. As illustrated in Fig. 13.1, almost half a year after its debut only 21.5% have installed the latest Android version and 7.5% use the latest update (version 8.1). While this may indicate that the vast majority of the users do not enjoy the new features of the platform, it also suggests that not all of its users have the same security updates installed.

Apart from the versioning fragmentation, we also have the variety of Android "flavours", also known as the different Android versions that manufacturers ship their devices with. The reason for these diverse flavours is the fact that manufacturers, despite following AOSP, they actually tailor the OS to their needs, most commonly by adding/removing apps and features to bind them with the firmware or by changing UI elements. Furthermore, it is also debatable whether all of the vendors follow the AOSP

[1] https://www.washingtonpost.com/news/wonk/wp/2013/06/12/heres-everything-we-know-about-prism-to-date/?utm/term=.2e201efd7097.

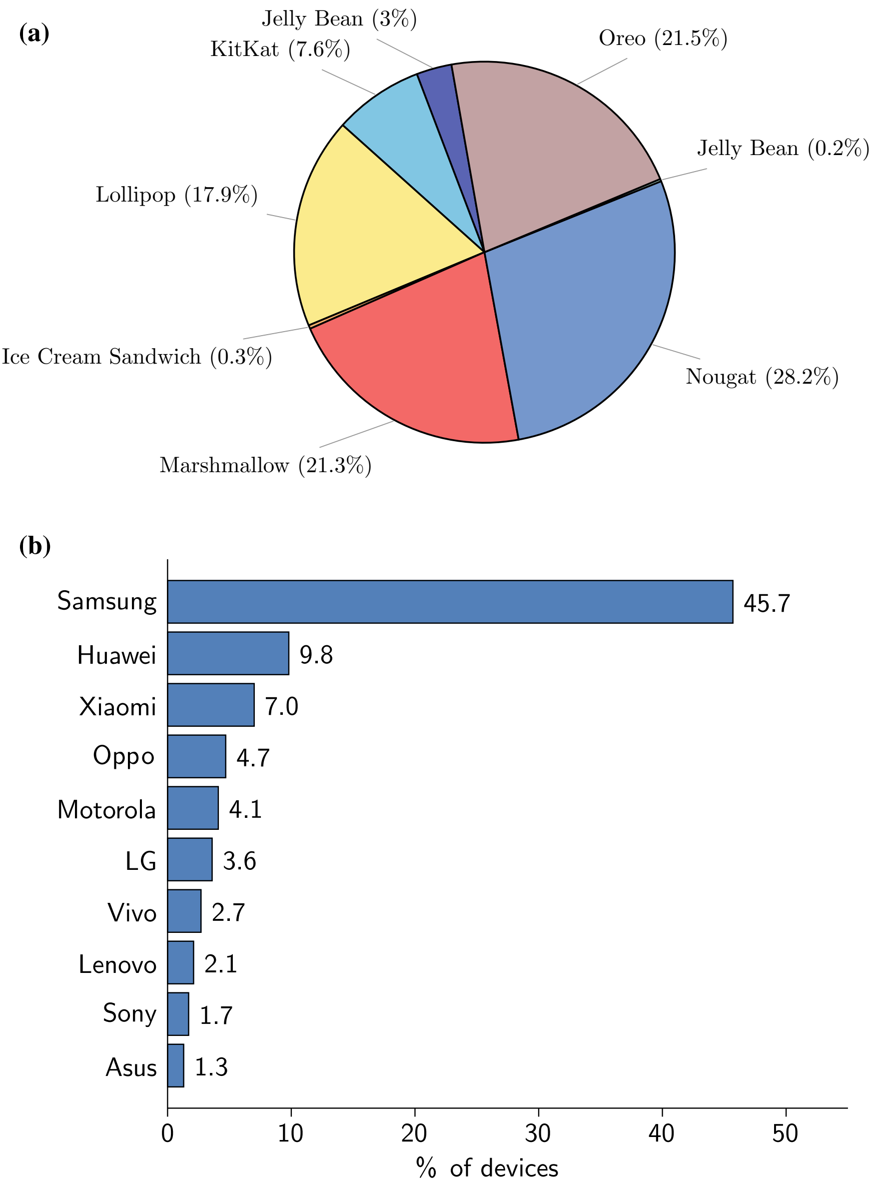


Fig. 13.1 **a** Android version statistics. *Source* Android Developer, **b** market share of Android manufacturers. *Source* Appbrain

guidelines regarding security and privacy requirements when they modify the original source code according to their needs. Google notifies them of already reported and fixed vulnerabilities in several versions. Since vulnerabilities are accompanied by Common Vulnerabilities and Exposures Identifiers (CVEs), it is easy to monitor whether a specific patch has been pushed into the device. Although major vendors usually report fixes of publicly known vulnerabilities,[2] updates not falling in line with the CVE system may be disregarded by vendors or even deliberately omitted if they happen to coincide with a provided functionality. Recently, using SnoopSnitch [20] it was discovered that not only Android vendors fail to provide security updates to their users or their updates are incomplete [17]. Finally, apps may target different API levels which implies different targetted functionality, but also different security standards. Note that the red vertical line indicates the API level 24 below which apps are vulnerable to our attacks, regardless of manufacturer and installed Android version.

13.1.2 Main Contributions

The main contribution of our paper is threefold. Firstly, we illustrate that a PRISM-like surveillance project collecting only communication metadata could be easily implemented through the distribution of seemingly benign apps that do not use any dangerous permissions. These apps could exploit privacy leakages partially fixed in AOSP, but not yet handled by the majority of smartphone vendors. More precisely, in this work, we demonstrate that by simply monitoring metadata available from the most well-known messaging apps in Android, "useful" and personal information about social interactions can be extracted by non-privileged apps. Table 13.1 contains the communication apps we examined. As a result, we show that the use of metadata bypasses the Android security mechanisms, allowing an adversary to extract a lot of sensitive information about individuals without requesting any dangerous permissions from the users. For instance, by simply exploiting inherent Android mechanisms, and without interacting with service providers or backdooring any communication app, one can determine with overwhelming probability not only when users interact with messaging apps, but also whether two users communicate with each other. The disclosure of this vulnerability aligns perfectly with the emerging belief that preventing privacy leakages in mobile environments is far more complex than one would expect [25].

Secondly, we discuss how the above approach can be used by unprivileged apps to derive usage statistics to build valuable user profiles, a case that would otherwise have required "system level" permissions. Beyond user profiling, these "monitoring" events can be further exploited by malware to timely interfere with user interaction resulting in a number of "unpleasant" situations, both for the users and also for the companies involved. Notably, Google is well aware of such threats; therefore with

[2] For instance see Samsung: https://security.samsungmobile.com/securityUpdate.smsb.

Table 13.1 Apps that are investigated and the reported installations according to Google Play

Application	Installations
BBM	100,000,000–500,000,000
Facebook messenger	1,000,000,000–5,000,000,000
ICQ	10,000,000–50,000,000
imo	100,000,000–500,000,000
KakaoTalk	100,000,000–500,000,000
LINE	100,000,000–500,000,000
QQ international	5,000,000–10,000,000
Signal	5,000,000–10,000,000
Skype	1,000,000,000–5,000,000,000
Snapchat	500,000,000–1,000,000,000
Telegram	100,000,000–500,000,000
Viber	500,000,000–1,000,000,000
WeChat	100,000,000–500,000,000
Wire	1,000,000–5,000,000
WhatsApp	1,000,000,000–5,000,000,000

the introduction of Nougat, apart from pushing many changes into the Android, they also tried to fix such leakages by further locking the contents of the corresponding `/data/data` directory. More precisely, as stated in Android 7.0 Behavior Changes:

> In order to improve the security of private files, the private directory of apps **targeting** Android 7.0 or higher has restricted access (0700). This setting prevents leakage of metadata of private files, such as their size or existence. [6]

Finally, our results indicate that these changes have not been implemented at all by some major manufacturers or they have been implemented partially by others, creating thereby a non-uniform and inconsistent landscape of different implementations with varying functionality, which exposes users' security and privacy.

13.1.3 Vulnerable Audience

Quantifying the vulnerable audience is rather complicated due to the difficulties in quantifying the constraints that have to be met to exploit the vulnerability. Therefore, it is easier to identify who is definitely secure. In this regard, users who are running AOSP since Nougat and whose applications are all running on API level above 23 are considered secure. In practice, this initially means that 70% of the devices are vulnerable as they are not running Nougat, hence the proper permissions have not been implemented. From the remaining 30% of the market we have no precise numbers, but qualitative data. First, most manufacturers have not implemented the “0700 policy", consequently their users are exposed. Second, even in the cases where some of the manufacturers have implemented this policy, the vast majority of apps does not target API levels above 23. Therefore, users of such applications are also exposed. Using Tacyt [11], we managed to identify developer trends since the beginning of the year, based on the target API levels of the updates they pushed to Google Play. We filtered the results to applications which have a tangible amount users, so our results refer to apps having more than 100 K downloads. For finance apps, developers pushed 38 (58%) of the 66 updates for secure API levels and 28 (42%) for insecure. Similarly, for communication apps 41 (57%) of the 71 updates was made for secure API levels and 30 (43%) for insecure. The above indicate that above 42% of the updates that are currently pushed for apps which by definition handle sensitive data are targeting insecure API levels. Notably, from all apps in Google Play having more than 100 K downloads which updated their APKs since the beginning of the year, 791 (39%) targeted insecure API levels, and 1229 (61%) targeted secure ones.

13.1.4 Organization of This Work

The rest of this work is organized as follows. In the next section, we provide an overview of Android internals related to our work. More precisely, we discuss Android app permissions and methods to derive the foreground app and usage statistics. Then, in Sect. 13.3 we discuss our metadata collection methodology which we use in Sect. 13.4 to showcase two different threat scenarios. In the first one, we illustrate how an unprivileged app can monitor the usage of the most well-known communication apps to derive social connections. In the second use case, we present a UI replication attack having Paypal as our reference. The article concludes in Sect. 13.5 where we summarize our contributions and findings and we propose remedies that could be applied in all the aforementioned attacks to mitigate user profiling and exploitation.

13.2 Related Work

13.2.1 Android Permissions

Although mobile devices may not compare with desktop computers in terms of computational capacities, they can provide an augmented user experience as they can quickly adapt to their context and interact with the user through various means due to the plethora of embedded sensors they are equipped with. However, many of these sensors such as microphone, camera and GPS can leak really sensitive information and therefore access to these resources is only permitted upon user consent. As of Android Marshmallow, the management of permissions has become more fine-grained, thus allowing users to grant and revoke consent whenever deemed necessary.

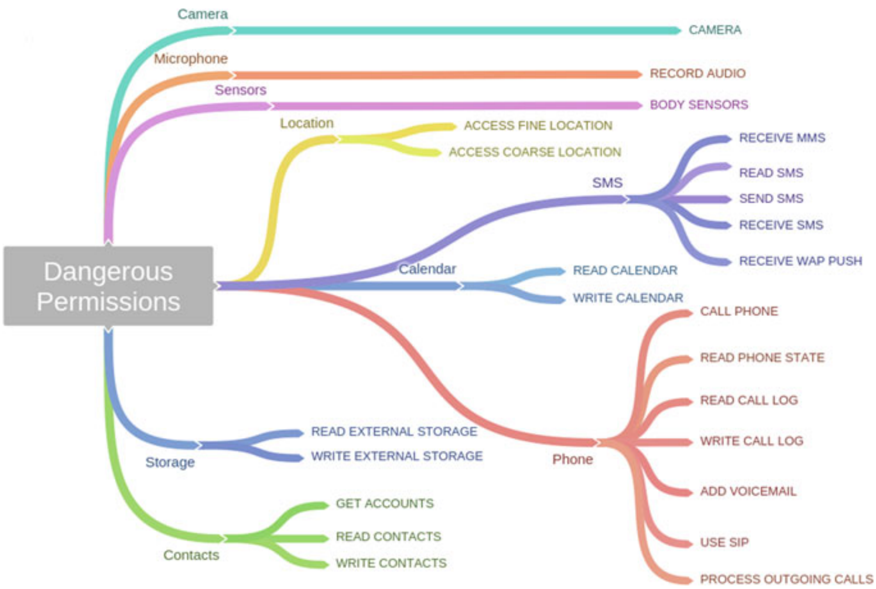


Fig. 13.2 Dangerous permissions in Android

Depending on the risk the users are exposed to, the permissions are mainly categorised as normal and dangerous. In this regard, normal permissions include, among others, access to the Internet, accelerometers and vibration while dangerous permissions include access to the microphone, camera, GPS, phone calls etc. The full list of dangerous permissions in Android is illustrated in Fig. 13.2. Apart from the level of the risk exposure involved, another significant difference between these two groups of permissions is in their management. While dangerous permissions can be revoked or granted whenever the user wants, normal ones are automatically granted once the app is installed and they cannot be revoked at all. Notably, if an app requires only normal permissions to be granted, the latest versions of Android prevent the user from even reading which these permissions are.

Apart from these two categories of permissions, Android has two more: Signature and SignatureOrSystem. The signature permission is designed for interoperability and enables applications which are signed with the same certificate to access the same resources even though only one of them is granted this access. SignatureOrSystem is a special permission designed for manufacturers to enable installing their applications in the Android system image and pertains to many elevated permissions such as rebooting the device or clearing caches. For a more detailed overview of Android permissions the interested reader may refer to [3].

To further protect the Android ecosystem, Google requires the user to explicitly grant some app permissions through completely different permission management screens (see Fig. 13.4b–d), which differ significantly from the traditional permission screen supplied for handling dangerous permissions (seen in Fig. 13.4a).

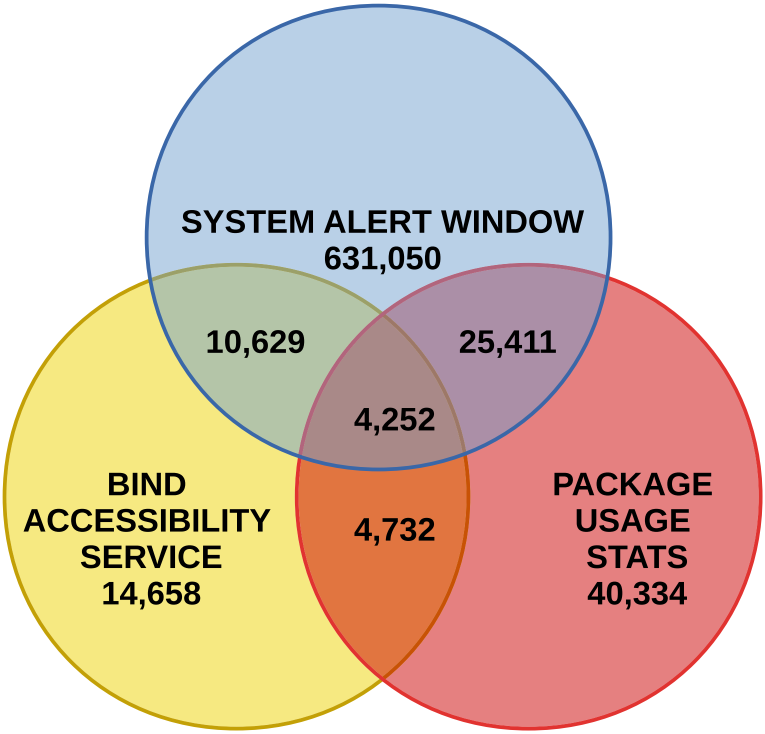


Fig. 13.3 Usage of some system permissions according to Tacyt. Numbers are reported in app versions

In the former category, we have permissions such as `SYSTEM_ALERT_WINDOW`, `BIND_ACCESSIBILITY_SERVICE`, `WRITE_SETTINGS` and `PACKAGE_USAGE_STATS`. To prevent users from carelessly granting these permissions, Android provides a completely different interface, and in principle, the corresponding settings are well-hidden in the menus so that users will grant access only when deemed necessary (see Fig. 13.4 for comparison). Nevertheless, these permissions are used by thousands of apps, as reported by Tacyt and seen in Fig. 13.3. To understand the extent of the risk that these permissions expose their users to, one has to consider that the `SYSTEM_ALERT_WINDOW` allows an application to overlay every Android activity and therefore can utterly deceive the user. The `BIND_ACCESSIBILITY_SERVICE` permission allows an application to imitate user tapping on the screen. Therefore, once granted to an app, it can perform any action on the user's device. Finally, the `PACKAGE_USAGE_STATS` will be discussed in detail in the following paragraphs.

Beyond the aforementioned categorisation of permissions, Android has the same Linux-based mechanism for UID/GID based access control. All users and groups are assigned with an ID (see Listing 1). As implied by this code excerpt, apps are

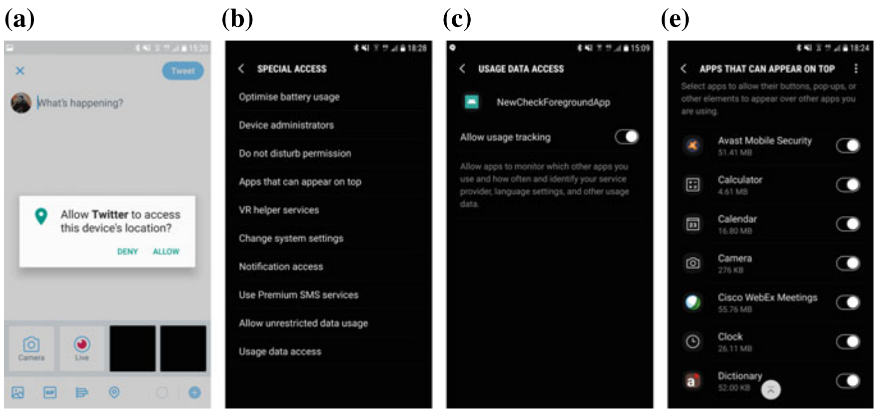


Fig. 13.4 Different interfaces for managing permissions in Android. **a** Granting a dangerous permission, **b** managing some system permissions, **c** allowing an app to track usage of other installed apps, **d** managing the permission to overlay other apps

assigned an ID above 10000, referred to as AID. Once a user grants a permission to an app, the app is added to the corresponding group, and it can access the particular resource. Therefore, if an app belongs to groups 1006 and 1021, it can access camera and GPS.

13.2.2 Android Foreground App

Android has an inherent problem regarding its UI: the lack of actual proofs of the identity of the application running in the foreground. This stems from two reasons, the size constraints of the devices which imply further constraints to UI, and also user permissions. According to the first one, UI components in Android, and mobile devices as a whole are stacked one on top of the other to fit into the small monitor of these devices. As a result, this size constraint prevents users from being able to determine the "actual" foreground app successfully. As for the second, users do not have many permissions, or even "high level" permissions, to determine themselves, or even by installing additional apps, the list of open apps and services in AOSP. Therefore, Android users blindly trust the Android UI.

In previous API levels, applications could monitor open apps, e.g. using the `getRunningTasks` method of `ActivityManager` as of API level 1. Nevertheless, as of API level 21, this method is no longer available to third-party applications. Google stated that:

> …the introduction of document-centric recents means it can leak person information to the caller.

```
#define AID_ROOT 0 /*traditional unix root user*/
/*The following are for LTP and should only be used for testing*/
#define AID_DAEMON 1 /*traditional unix daemon owner*/
#define AID_BIN 2  /*traditional unix binaries owner*/
#define AID_SYSTEM 1000 /*system server*/
#define AID_RADIO 1001 /*telephony subsystem, RIL*/
#define AID_BLUETOOTH 1002 /*bluetooth subsystem*/
#define AID_GRAPHICS 1003 /*graphics devices*/
#define AID_INPUT 1004 /*input devices*/
#define AID_AUDIO 1005 /*audio devices*/
#define AID_CAMERA 1006 /*camera devices*/
#define AID_LOG 1007 /*log devices*/
#define AID_COMPASS 1008 /*compass device*/
#define AID_MOUNT 1009 /*mountd socket*/
#define AID_WIFI 1010 /*wifi subsystem*/
#define AID_ADB 1011 /*android debug bridge (adbd)*/
#define AID_INSTALL 1012 /*group for installing packages*/
#define AID_MEDIA 1013 /*mediaserver process*/
#define AID_DHCP 1014 /*dhcp client*/
#define AID_SDCARD_RW 1015 /*external storage write access*/
#define AID_VPN 1016 /*vpn system*/
#define AID_KEYSTORE 1017 /*keystore subsystem*/
#define AID_USB 1018 /*USB devices*/
#define AID_DRM 1019 /*DRM server*/
#define AID_MDNSR 1020 /*MulticastDNSResponder (service discovery)*/
#define AID_GPS 1021 /*GPS daemon*/
#define AID_UNUSED1 1022 /*deprecated, DO NOT USE*/
...
#define AID_APP 10000 /*TODO: switch users over to AID_APP_START*/
#define AID_APP_START 10000 /*first app user*/
#define AID_APP_END 19999 /*last app user*/
    ...
```

Listing 13.1: Excerpt from available UIDs/GIDs in Android as defined in AOSP source code [15].

However, security researchers managed to derive the foreground apps through leaks from the `procfs`, as Android, like all Linux-based operating systems, uses it to store information of the processes that are executed by the OS.

Towards this end, Chen et al. [10] monitored offline the memory consumption of each activity in an application by tracking the memory allocation of the corresponding `/proc/[pid]/statm` file. They hypothesize that there is a specific footprint when shifting from one activity to another which can be used to identify app and activities. The latter can be further improved by monitoring network traffic through `/proc/net/tcp6`.

Bianchi et al. in [8] also identify the foreground application by using `procfs`. In this case, the leakage is from the file `/proc/[pid]/cgroups` whose contents change from `/apps/bg_non_interactive` to `/apps` when an app is sent to the foreground.

Finally, Alepis and Patsakis [2] exploited the `oom_adj_score` file in `procfs`, a file used by Android to monitor resource allocation and release. Depending on app usage, Android modifies this file which is stored under the directory `/proc/[pid]/` of each app. By pruning all the system applications, the least likely process to be killed is the foreground app.

As of Android Nougat, access to `/proc/[pid]/` is prohibited to other apps; therefore all the aforementioned attacks are not applicable to the two latest Android versions. However, as of Android Lollipop, developers may use two additional methods to accomplish foreground app detection. Namely, either through the utilization of the `UsageStatsManager` API which requires the `PACKAGE_USAGE_STATS` permission and allows an app to collect statistics about the usage of the installed apps, or through the `AccessibilityService` API which requires the `BIND_ACCESSIBILITY_SERVICE` permission. Regarding the first case, this kind of information is presumably vital for the Android ecosystem, considering that it requires system permission to be collected. Yet, Android does not intend to allow an app to derive anything else apart from aggregated statistics about the usage of the installed apps. Therefore, the app can get statistics but they are not very fine-grained: the app can collect aggregated usage data for up to 7 days for daily intervals, up to 4 weeks for weekly intervals, up to 6 months for monthly intervals, and finally up to 2 years for yearly intervals, always depending on the chosen interval. Nevertheless, this service gives developers the ability to build a list of usage statistics per app, `List<UsageStats>`, and consequently query each of its items through the built-in `getLastTimeUsed()` method, to derive the foreground app. The second option, utilizing the AccessibilityService, includes handling the `onAccessibilityEvent()` callback and checking whether the `TYPE_WINDOW_STATE_CHANGED` event type is present, to determine when the current window changes. Finally, the target windows is further checked to determine whether it is the case of an activity by the method `PackageManager.getActivityInfo()` and the foreground app is revealed. It is important to note that Google has warned developers about this permission, that she will remove apps from the Play Store if they use accessibility services for "non-accessibility purposes" [24].

In the following paragraphs, we discuss in detail why app usage statistic data and also foreground application detection are actually considered such sensitive user information, using concrete examples of their malicious usage.

13.3 Collecting App Metadata

13.3.1 Assumptions and Desiderata

In our threat model, we assume that the victim has been tricked into installing a malicious app in his Android device. This assumption is considered standard in most Android related attacks [12, 13, 27]. Therefore it is aligned with the current literature. To relax possible constraints, we further assume that the device is not rooted; therefore, the attack could be launched in every stock Android installation. Finally, we assume that the application does not request any dangerous permissions from the user. The latter is rather crucial as dangerous permissions require further run-time user interaction in OS versions following Android Marshmallow and they can also be revoked later, as already discussed. Moreover, dangerous permissions can also deter users from installing an app.

Regarding code and library dependencies, we assume that the malicious app, apart from listing the corresponding app files, it does not request any additional shared library and does not make any suspicious API calls. To hide the actual filenames from possible static code analysis, we consider that the adversary collects them on runtime through a remote server. To this end, the app can use the Firebase or Azure cloud infrastructure that would be utilized for all regular interaction tunnel all of its traffic through legitimate servers, as used in [18] by social botnets, and hence to hide its communication with the C&C server.

Having a "zero permission" app, that is an app requesting only normal permissions, which communicates only with the Google servers not only makes the app look benign for the user, but it also bypasses many static security controls such as permissions, API calls, and network connections, that many consider as key indicators for identifying malicious apps [1, 7, 16, 23, 28, 30].

Finally, we make two weak assumptions to monitor whether two or more users are communicating via a specific app. The first one is that the users are simultaneously online, and thus there is no significant delay in message delivery between apps. This is a common case when users are interacting with "instant" messaging apps since by default these apps are used in real-time mode. The second weak assumption is that both apps are synchronised with a remote clock for precise timing. The latter is required for preventing errors due to time lags and for detecting whether user A contacted user B or the other way round. This can also be achieved by sending instant reports to the server once an activity has been recorded.

13.3.2 Basic Concept

While Android apps are isolated from each other, as they belong to different users, specific metadata can be extracted from their corresponding files. As the underlying filesystem of Android is `ext4` [19], as in many Linux installations, the file permissions are also similar to other Linux systems. Practically, depending on the user permissions, a user is allowed to (r)ead, (w)rite or e(x)ecute a file. Therefore, listing the contents of a directory or a file depends on the already assigned user permissions. All user installed apps in Android are installed in a directory `/data/data/<package_name>`, where `package_name` stands for the name of the installed app's package, and it is of the form `com.xyz`.

As in any system, misconfiguration of file permissions can leak much sensitive information which may lead to full compromise. More specifically, apps store their different "types" of stored data in corresponding subdirectories:

- databases/: Storage for the app's databases
- lib/: Storage for libraries and app helpers

- files/: Storage for app related files
- shared_prefs/: Storage for shared preferences and usually app settings
- cache/: Storage for caches

If an app does not want to share specific information with other apps, the contents of the underlying files and directories under folder `/data/data/com.xyz` are by default inaccessible by other apps, hence users. Nevertheless, if the permissions of the corresponding files and folders are not properly set and "someone" knows the absolute location of a file, even though he may not be able to access the contents of the file, he might be able to derive evidence of its existence, or even metadata about it. Typically, in Linux-based systems, this can be performed via various commands like:

```
ls -l /data/data/com.xyz/secret_file
stat /data/data/com.xyz/secret_file
```

Both commands, among other data, contain the last modification time and size of the file. Clearly, this seems only a small piece of information which can be collected by any installed app in Android without requesting any permission from the user, not even the most profound one: "Storage". Correspondingly, the research question in stake is whether this small leak of information can be used to derive sensitive information about the user. As already discussed, Google as of Android Nougat decided to remove all possible access from these files by setting the file permissions to 0700. Nevertheless, this change was not assigned to any CVE and, to the best of our knowledge, app permissions in file level do not follow a specific "default" pattern. In fact, throughout our research, we found different permission patterns in app files, discussed in more detail in the next section. It is worthy to notice that even if a CVE had been assigned and a patch had been made available, as recently shown by Lell and Nohl [17], manufacturers may often lie about their integration. Moreover, applications do not follow a specific pattern in the way they store their data. Even though, as already discussed, there are directories regarding libraries, caches, shared preferences, databases and file storage, inside these "default" directories, there is no specific rule to be followed. As a result, in many occasions related to data storage, apps have their data spread among different local databases, most often SQLite. Therefore, different user interactions correspond to changing different files which can then be traced to reveal the interaction.

The obvious drawback of the aforementioned issues is that many vendors could leak which applications are currently running or even the kind of interaction they are performing. Although running applications can be considered as trivial information, it is a considerably critical piece of information for the Android ecosystem since, apparently, by monitoring running applications one can derive usage statistics, vital information for user profiling and mobile targeted advertisement. In the most sinister scenario, if one can derive which is the foreground app, he can easily proceed in tricking the users into disclosing their credentials by pushing forward a forged Android activity that imitates the UI of the user-initiated app, running in the foreground [2, 8, 10, 22].

According to the official source [5], in Android Nougat, API level 24 and above, the private directory of apps has restricted access to improve the security of private files. As a result, as of API level 24, apps cannot access the `/proc/PID` directory for other PIDs, rendering all relevant to the "proc" directory approaches to detect foreground apps useless. Nevertheless, as we are illustrating in this paper, other directories and specific file metadata enable us to achieve the same result in the vast majority of Android smartphones to date.

13.3.3 Methodology

We opted for a black box methodology to see the problem in its full extent. Therefore, our goal was to determine how much information can be collected from an adversary who does not have any access to the apps' internals.

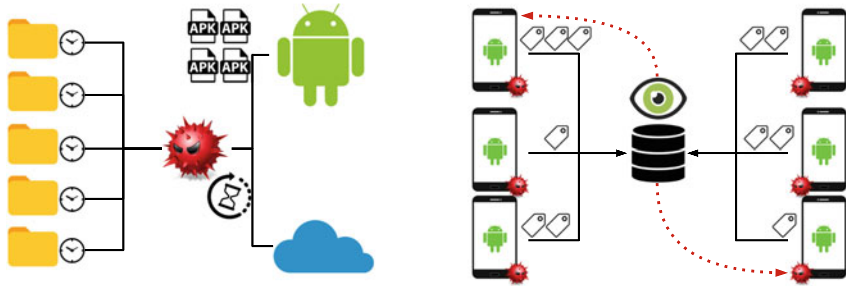
To bypass the restrictions of file access permissions in AOSP, we started our experiments with two devices, a rooted smartphone and one using AOSP. First, we installed the same apps and versions in both devices to have a common reference point, and we created fake user profiles for each application when deemed necessary. Then, we went through each application independently and started interacting with it, keeping track in the rooted smartphone of which files were changing upon every action. In this regard, the rooted phone can be considered as the "file change sensor" module, while the non-rooted phone provided the necessary "triggers" to initiate specific user actions.

Finally, after having noted all the involved files for the metadata exploitation, we implemented an Android app which acts as a background service. The service silently monitors in the background the metadata of specific files of the installed apps and subsequently transmits any detected changes to a remote database. Apart from this passive role, the app is also capable of presenting an Android activity whose UI is rendered from data that it fetches from the database when the proper trigger is sent.

The basic concept is illustrated in Fig. 13.5. Once the user installs the malicious app X, it retrieves the specs of the device and the Android OS version and a list of all installed apps from Android's Package Manager system service. X communicates this list to its server to obtain a list of files that it should monitor. Once received, X running as a background service, monitors the aforementioned files at predetermined intervals to derive the last modification time and size, which are then sent to the C&C server (see Fig. 13.5a).

To efficiently monitor the filesystem changes in every directory recursively, we used the `inotifywait` utility. The events that we monitored were: *create*, *delete*, *attrib* and *modify*. Since `inotify` cannot be directly installed in Android, we exploited the features of a well-known terminal emulator for Android devices, Termux.[3] The feature of Termux that we aimed is the `apt` integration that allows the user to install Linux packages, in our case `inotify-tools`.

[3] https://termux.com/.



(a) Collecting local information. The malicious app periodically checks the modification time of specific files.

(b) Correlating information from many devices in the C&C server and pushing commands to specific devices.

Fig. 13.5 Basic concept

13.4 Experimental Results

As discussed, Google introduced in AOSP the 0700 permissions in Nougat, which at the moment of writing means that officially 70% of all Android devices are vulnerable. However, the number of the actually affected devices is, alarmingly, far bigger. In our experiments we tested several devices from top major Android vendors running Android API level$\geq$ 24, that is devices running Nougat and Oreo. More precisely, we experimented with devices from Samsung, LG, Xiaomi, Huawei and HTC. The “pure” Android devices (Nexus and Pixel) were the only ones that did not present any metadata leakages. On the contrary, devices which did not run pure Android, even if they were from the same manufacturer, leaked file metadata. To leak this information, we used the native `os.stat` method for each file we identified.

In what follows we present three different threat scenarios that stem from the described metadata leakage. In the first two cases, we focus on surveillance, while in the latter we focus on harvesting credentials.

13.4.1 Monitoring the Android Filesystem

In this scenario we try to derive usage statistics from the OS regarding the following actions:

- Add/Remove a user to contacts.
- Send/receive an SMS.
- Make/receive a call.
- Enable/disable GPS
- Shoot a photo.

Table 13.2 Identified Android actions

	Version	Call		Message		GPS		Contact		Camera
		Make	Receive	Send	Receive	On	Off	Add	Delete	
HMD Global TA-1024 (Nokia 5)	8.1.0			✓	✓	✓	✓			
Huawei Nexus 6P	8.1.0	✓	✓	✓	✓	✓	✓	✓	✓	✓
LGE LG-H870	8.0.0			✓	✓	✓	✓			
Xiaomi Mi A1	8.0.0			✓	✓	✓	✓			
Xiaomi MI 6	8.0.0	✓	✓	✓	✓	✓	✓			
HUAWEI VTR-L09	8.0.0			✓	✓	✓	✓			
Xiaomi Redmi 4A	7.1.2	✓	✓	✓	✓	✓	✓	✓	✓	
Xiaomi Redmi Note 5A Prime	7.1.2		✓	✓	✓	✓	✓			
Samsung SM-J510FN	7.1.1	✓	✓	✓	✓	✓	✓	✓	✓	
Xiaomi Redmi Note 4	7.0			✓	✓	✓	✓	✓		
Samsung SM-G930F	7.0			✓	✓	✓	✓	✓	✓	
Samsung SM-G935F	7.0			✓	✓	✓	✓	✓	✓	
Samsung SM-J710F	7.0			✓	✓	✓	✓	✓	✓	
HUAWEI PRA-LX1	7.0	✓	✓	✓	✓					
Samsung S6	7.0			✓	✓	✓	✓	✓		
Samsung SM-A500FU	6.0.1	✓	✓	✓	✓	✓	✓	✓	✓	
Xiaomi Redmi Note 3	6.0.1			✓	✓			✓	✓	✓
Xiaomi Redmi Note 4	6.0			✓	✓	✓	✓	✓	✓	
Plaisio Computers SA Turbo-X_A2	6.0			✓	✓	✓	✓	✓	✓	✓
Huawei P9 lite	6.0	✓	✓	✓	✓	✓	✓			
HUAWEI ALE-L21	6.0			✓	✓	✓	✓			✓
Sony D2303	5.1.1			✓	✓	✓	✓	✓	✓	
Motorola XT1032	5.1			✓	✓	✓	✓	✓	✓	
Samsung SM-J320FN	5.1.1			✓	✓	✓	✓			✓
Samsung SM-J320F	5.1.1			✓	✓	✓	✓	✓	✓	
HTC One	5.0.2			✓	✓	✓		✓	✓	✓

without requesting any permission from the user, just by monitoring filesystem changes. To this end, we monitored a series of files of Android, as seen in Table 13.4, to determine the aforementioned actions. While several variations are depending on the underlying Android flavour, it is evident that an adversary can easily determine a wide set of actions as illustrated in Table 13.2.

13.4.2 Communication Apps

In this threat scenario, we try to derive usage statistics from communication apps. Since locality plays a crucial role in their usage (see Fig. 13.6), we studied the most well-known, secure and widely used communication apps in Google Play. The list (see Table 13.1) contains 15 apps, all of which have at least one million installations while their vast majority has more than 100 million downloads.

In our experiments, we aimed to detect more than the fact of using the apps. To this end, we wanted to determine whether we could identify users communicating with each other, when and how. Using the methodology detailed in the previous section, we performed the following actions in each app and device:

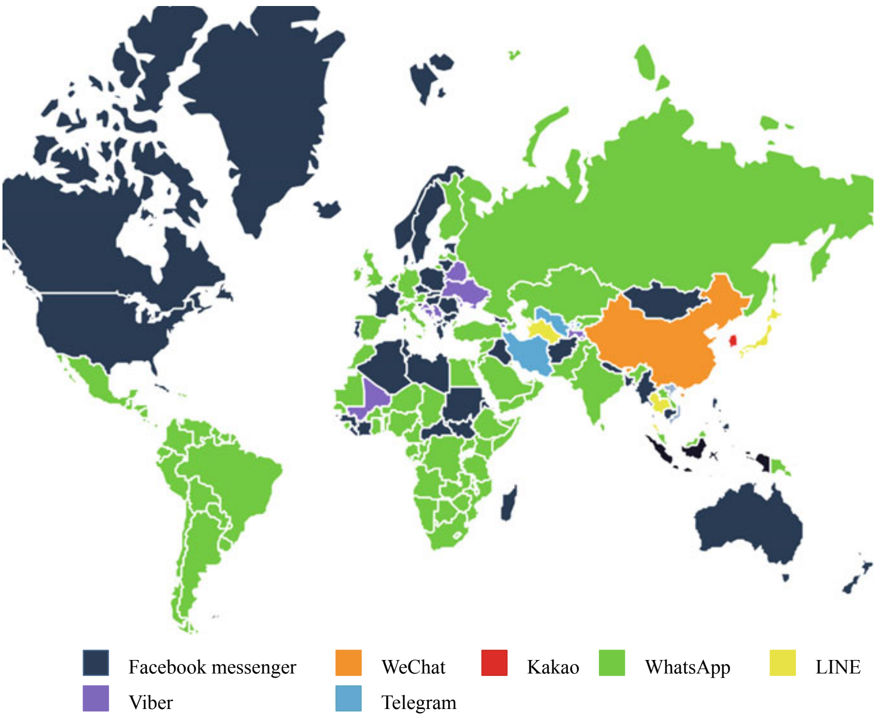


Fig. 13.6 Most popular apps per country. *Source* SimilarWeb

- Add a user to contacts.
- Remove/Ban a user from contacts.
- Send a text message.
- Initiate a call.
- Initiate a video call.

The above actions were performed multiple times from each device to determine the files affected each time, how fast these changes were made, and to prune any coincidental file changes. It is worthwhile to note that most of these apps would periodically change their files as they communicate with the corresponding service and receive some status messages. While some of these changes created noise in our samples, they were easy to be pruned based on simultaneous similar changes made to other installed apps. Apparently, the changes were initially monitored only in the rooted device. Then, knowing the locations of the altered files, we validated our results with the non-rooted device too.

The results of our experiments are illustrated in Table 13.3. From the 15 apps, only 6 were secure (their row is highlighted in gray), meaning that no metadata could be extracted from them. Then, for three more we could detect that some app interaction occurred, yet it could not be precisely identified ("Unidentified Action" column in Table 13.3). The reason for the latter is that some apps used a small number of databases, one or two. Thus, every interaction was mapped to the same files, not giving us the opportunity to track "specific" actions. Yet, the most interesting results are for the rest 6 apps, most of which have users on the scale of billions.

More precisely, for Facebook messenger, we can determine when the user blocks someone. Moreover, we can ascertain when the user initiates a communication with another user. While we cannot determine whether this occurs via a message or a call, we can trace the communication and determine from the respective timestamps who is sending and who is receiving if both devices have the monitoring app installed. In the case of Line, we can only detect when the user receives a message. In Skype we can tell that a user has performed one of the following actions: add/block a contact, send a message or receive a call. In the case of Snapchat, we can determine when a user receives or sends messages and, if both communicating users have installed the monitoring app, we can determine from the timestamps which one is the sender and the receiver. Additionally, we are able to trace an interaction which is either the addition/removal of a contact or the sending/receipt of a snap. We argue that if both parties have installed our monitoring app, the success rate of our approach is further increased. For Wire we can determine when the user adds a new contact and when the user interacts with others, but without being able to determine the mean (distinguish between messages and calls). Finally, in the case of WhatsApp, we can determine when a user initiates a phone call. However, we cannot determine exact message receipt/sending, or call received in the same device. Again, if both parties have installed our monitoring app, these results can be further fine-tuned.

The full list of the affected files for each application is illustrated in Table 13.4. Figure 13.7 illustrates part of the problem in terms of file permissions. Files and folders of WhatsApp have different file permissions, allowing an adversary to extract a lot of metadata about them.

Fig. 13.7 Partial structure and permissions of the folder that Whatsapp is installed as recorded in the rooted phone

```
/
└ data
  └ data
    ├ ...
    ├ com.whatsapp
    │ ├ cache (771)
    │ │ └ com.android.opengl.shaders_cache
    │ ├ code_cache (700)
    │ │ └ secondary-dexes (700)
    │ │   └ MultiDex.lock (600)
    │ ├ databases (771)
    │ │ ├ _jobqueue-WhatsAppJobManager (660)
    │ │ ├ axolotl.db (660)
    │ │ ├ chatsettings.db (660)
    │ │ ├ location.db (660)
    │ │ ├ msgstore.db (660)
    │ │ └ wa.db
    │ ├ files (771)
    │ │ ├ key (600)
    │ │ ├ me (660)
    │ │ ├ rc2 (600)
    │ │ ├ statistics (600)
    │ │ └ Logs (700)
    │ │   └ whatsapp.log (600)
    │ ├ lib (755)
    │ │ └ no_backup (700)
    │ │   └ com.google.android.gms.appid-no-backup (600)
    │ └ shared_prefs (771)
    │   ├ com.google.android.gms.appid.xml (660)
    │   ├ keystore.xml (660)
    │   ├ multydex.version.xml (660)
    │   └ registration.RegisterPhone.xml (660)
    └ ...
```

Based on the above, it is apparent that unauthorized apps can derive a lot of valuable information from communication apps such as when they have been used and, should the monitoring app has been installed in both devices, to whom is interacting with whom, how and when.

Table 13.3 Actions that can be deduced from metadata per app. Cells marked with $^{+}$ in the same row denote that an action has been identified and can be narrowed down to one of them. Same applies for *. Snapchat does not enable calls, but “snaps”

	Contact			Message		Call		Unidentified
	Add	**Delete**	**Block**	**Send**	**Receive**	**Make**	**Receive**	**Action**
BBM								
Facebook messenger			$\checkmark$	$\checkmark^{+}$	$\checkmark^{+}$	$\checkmark^{+}$	$\checkmark^{+}$	
ICQ								$\checkmark$
imo								$\checkmark$
Line					$\checkmark$			
KakaoTalk								$\checkmark$
QQ International								
Signal								
Skype	$\checkmark^{+}$		$\checkmark^{+}$	$\checkmark^{+}$			$\checkmark^{+}$	
Snapchat	$\checkmark^{+}$	$\checkmark^{+}$		$\checkmark^{*}$	$\checkmark^{*}$	$\checkmark^{+}$	$\checkmark^{+}$	
Telegram								
Viber								
WeChat								
Wire	$\checkmark$			$\checkmark^{+}$	$\checkmark^{+}$	$\checkmark^{+}$	$\checkmark^{+}$	
WhatsApp				$\checkmark^{+}$	$\checkmark^{+}$	$\checkmark$	$\checkmark^{+}$	

13.4.3 Forged UI

Currently, there exist malware targeting bank applications which use overlays and UI replication of numerous banking applications.[4] Malware of this kind include families such as Bankbot, Bankun, Koler and SlemBunk, while there are other malware families like ransomware, e.g. Lockdroid, which also exploit these capacities. MazarBot and Svpeng Malware are two malware families which are even worse as they manage, after obtaining administrator privileges on the infected device, to trick users to give away their banking credentials. Moreover, new variants of Svpeng have keylogger capacities thanks to Accessibility Services. To quantify the problem, it must be highlighted that, according to CheckPoint [9], 74% of ransomware, 57% of adware, and 14% of banker malware abuse the notorious `SYSTEM_ALERT_WINDOW` permission.

While there are plenty of ways for the aforementioned malware to overlay the actual UI such as those described in [2, 4, 14, 29], the key ingredient of all is to timely present the overlay to the user. As discussed, these methods cannot be applied for API levels >23; therefore, other methods could be used to trick the user into disclosing the foreground app, e.g. forged shortcuts and notifications [2, 22]. However, since apps can retrieve the list of installed apps, they could try to use the metadata to infer when a targeted app is used.

To showcase the issue, we used Paypal as our reference. Using the same methodology, we monitored which files of Paypal are changed once the user logs in or logs out. The exact list of `data/data/com.paypal.android.p2pmobile/` direc-

[4] https://blog.avast.com/mobile-banking-trojan-sneaks-into-google-play-targeting-wells-fargo-chase-and-citibank-customers.

tory is illustrated in Listing 2. Therefore, an attacker can easily determine when the user signs in to Paypal.

The attack is now straightforward: an app that requests only normal permissions monitors the Paypal's target file. When the user logs in, the malicious app comes to the foreground, replicating the UI of the Paypal app. To prevent detection, the app can collect its data regarding the fake UI construction during runtime from the Internet and replicate the UI through a web form that is displayed within a web view. Since the user was already using the app, he considers that there was an error in typing his credentials and types them again, without knowing that they are disclosed to the malicious app. Once collected, the credentials are sent to the C&C server, and the app launches Paypal to allow the user to interact with the right app. The concept is illustrated in Fig. 13.8. Note that in this scenario the malicious app does not need any special permission to overlay Paypal as it uses UI replication and the forged UI completely covers the genuine one. Its only requirement is to come to the foreground, something that all apps in Android have the permission to accomplish.

```
shared_prefs/PresentationAccount.RememberedUserState.xml
shared_prefs/FoundationAccount.AccountState.xml
shared_prefs/FoundationCore.DeviceInfoState.xml
shared_prefs/version.6.shared.keys.xml
files/com.paypal.android.AccountInfo.secure
files/CoreStateData
files/AdjustIoActivityState
```

Listing 13.2: Files changed when a user logs in to Paypal.

13.4.4 User Metadata Experiment

In our independent research, to determine the extent of the problem we have also conducted a supplementary experiment involving distinctly statistical user data. More precisely, we created an app that collects both hardware and software specs from users' devices, accompanied by a test about the metadata leakage in question. The resulting data bundle from each user was sent to our server for further processing. This limited yet enlightening experiment justified our claims, about Android fragmentation that results in vendors failing to conform to AOSP security guidelines, while the same is true about app developers too. More precisely, our short-term experiment involved 120 random users. The analysis of the results showed that almost half of them, namely 56 users (46%) owned an Android smartphone with an OS API level beyond 23 (Marshmallow). This already means that 54% of the users in our evaluation were already vulnerable to the metadata leakage. In the remaining 46%,

Table 13.4 Files affected by each action per application

	Contact				Message	Call	
	Add	**Delete**	**Block**	**Send**	**Receive**	**Make**	**Receive**
Facebook messenger /data/data/com.facebook.orca/databases/			prefs_db graphql_cache threads_db2	prefs_db threads_db2	prefs_db threads_db2	prefs_db threads_db2	prefs_db threads_db2
imo /data/data/com.imo.android.imoim/				databases/imofriends.db	databases/imofriends.db	databases/imofriends.db	databases/imofriends.db
kakaoTalk /data/data/com.kakao.talk/databases/				KakaoTalk.db	KakaoTalk.db	KakaoTalk.db	KakaoTalk.db
Line /data/data/jp.naver.line.android/databases/					naver_line naver_line_push_history		
Snapchat /data/data/com.snapchat.android/databases,	com.snapchat.android.analytics.framework com.snapchat.android.analytics.frameworkshadow tcspahn.db unlockable_encrypted_database.db	com.snapchat.android.analytics.framework com.snapchat.android.analytics.frameworkshadow tcspahn.db unlockable_encrypted_database.db		com.snapchat.android.analytics.framework com.snapchat.android.analytics.frameworkshadow tcspahn.db	com.snapchat.android.analytics.framework com.snapchat.android.analytics.frameworkshadow tcspahn.db	com.snapchat.android.analytics.framework com.snapchat.android.analytics.frameworkshadow tcspahn.db unlockable_encrypted_database.db	com.snapchat.android.analytics.framework com.snapchat.android.analytics.frameworkshadow tcspahn.db unlockable_encrypted_database.db
WhatsApp /data/data/com.whatsapp/databases/				msgstore.db-wal msgstore.db-shm axolotl.db-shm axolotl.db-wal	msgstore.db-wal msgstore.db-shm axolotl.db-shm axolotl.db-wal		msgstore.db-wal msgstore.db-shm axolotl.db-shm axolotl.db-wal
Wire /data/data/com.wire/databases/				mixpanel ZGlobal.db-shm ZGlobal.db-wal evernote_jobs.db	mixpanel ZGlobal.db-shm ZGlobal.db-wal evernote_jobs.db	mixpanel ZGlobal.db-shm ZGlobal.db-wal evernote_jobs.db	mixpanel ZGlobal.db-shm ZGlobal.db-wal evernote_jobs.db

Fig. 13.8 Overlaying Paypal interface with forged interface

we further examined whether the vendors of the devices had incorporated the file permission security update which was introduced in Nougat.

The results showed that two major vendors, (see Fig. 13.1), namely Samsung and Xiaomi, do not meet the criteria, as well as OnePlus. For these three well-known manufacturers, we have discovered metadata leakage in popular messaging applications, which, most importantly, have a target API level (app specific targetSdkVersion) above 23. Regarding the case of Samsung, our experiment identified five of the most popular device models that are vulnerable, while we have found two popular models from Xiaomi and one from for OnePlus. Of course, these findings indicate that the problem is generic regarding the vendors and not specific devices. However, this argument is subject to the extent of the experiment.

Going one step further we have examined the number of vulnerable devices where the API level of Android is beyond 23 (Nougat and Oreo), yet the target app for the metadata leakage has a targetSdkVersion below 24. The results are quite alarming since we did not find any protected device in this case. Namely, the vendors that are affected by this vulnerability in our study are Samsung, Huawei, Xiaomi, LGE Nexus, LG, OnePlus, Sony and HMD Global (NOKIA). It is important to note that for these vendors, Google's models (Nexus/Pixel) are also included, while a significant number of these devices were running Android Oreo.

Concluding, our reported metadata leakage issue affects end-users in an overwhelming majority of cases since all parties involved can be found to be defective in the vulnerability in question. Summarizing, the OS is vulnerable in all versions prior to Nougat. Even in the cases after Nougat, device manufacturers may provide users with "insecure" devices by not following the AOSP guidelines. Even after Nougat and with conformance to the AOSP guidelines, app developers, including top companies (e.g. Facebook apps like Facebook, Facebook Messenger, and Instagram) may provide users with "insecure" apps, by not targeting their app API beyond 23.

13.4.5 Proofs of Concept

To validate the results of our work, we provide two deliberately stub implementations of the attacks. The first APK monitors the aforementioned communication apps and the second one the login actions to Paypal. All recorded actions are displayed as a log at https://monitor1webapp.azurewebsites.net/. To this end, we record a timestamp, the UserID (in this case AndroidID), the application which was being monitored and the action which was detected.

13.5 Conclusions

In this paper we have provided evidence that metadata leakage in not only existent, but in can easily lead to user data harvesting, user profiling and even user surveillance, impinging on users' lives. After analyzing the results of this study, it seems quite straightforward that the manufacturers need to fully embrace AOSP to implement security requirements effectively into their devices. However, at the time of writing this scenario is not the case, neither seems realistic or feasible given the huge deal of effort that would admittedly require. Therefore, the solution probably lies in the developers' hands. A potential patch that could be easily deployed and would help towards a solution of the "metadata leakage" problem for each app would be to utilize app subfolders with random names. To do this, the app upon installation or update should not be assigned to any local database. Instead, upon its first execution, it should create a folder using a nonce for its name, and it should store this name localy, e.g. in its shared preferences. Thus, it would be impossible for a malicious app to derive the folder name without exploiting a major app-specific vulnerability. Consequently, the patch would manage its goal since the attack stems from the fact that the apps' folder names are fixed and consequently "expected", with wrong permissions assigned to them that the apps cannot change. Obviously, to implement the above, the requested changes would only involve changing the connection string to include the prefix in the corresponding files along with adding the necessary queries to generate the initial structure in local databases. Likewise, one could also rename the corresponding files by adding a prefix that would be also stored in shared preferences, given that apps do not have access to the shared preferences of their peers unless they have root privileges.

Acknowledgements The authors would like to thank *ElevenPaths* for their valuable feedback and granting them access to Tacyt.

References

1. Y. Aafer, W. Du, H. Yin, Droidapiminer: mining API-level features for robust malware detection in android, in *International Conference on Security and Privacy in Communication Systems* (Springer, 2013), pp. 86–103
2. E. Alepis, C. Patsakis, Trapped by the UI: the android case, in *International Symposium on Research in Attacks, Intrusions, and Defenses* (Springer, 2017), pp. 334–354
3. E. Alepis, C. Patsakis, *Unravelling security issues of runtime permissions in android* (J. Hardw. Syst, Secur, 2018)
4. Y. Amit, Accessibility clickjacking the next evolution in android malware that impacts more than 500 million devices (2016). https://www.skycure.com/blog/accessibility-clickjacking/
5. Android Developer, Permission changes. https://developer.android.com/about/versions/nougat/android-7.0-changes.html. Accessed 07 Feb 2018
6. Android Developer. Android 7.0 behavior changes (2017). https://developer.android.com/about/versions/nougat/android-7.0-changes.html
7. D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, C.E.R.T. Siemens, Drebin: effective and explainable detection of android malware in your pocket. NDSS **14**, 23–26 (2014)
8. A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, G. Vigna, What the app is that? deception and countermeasures in the android user interface, in *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (IEEE Computer Society, 2015), pp. 931–948
9. Check Point Mobile Research Team, Android permission security flaw (2017). https://blog.checkpoint.com/2017/05/09/android-permission-security-flaw/. Accessed 09 Sep 2017
10. Q.A. Chen, Z. Qian, Z.M. Mao, *Peeking into your app without actually seeing it: UI state inference and novel android attacks, in 23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, USENIX Association, 2014), pp. 1037–1052
11. ElevenPaths, An innovative tool for the monitoring and analysis of mobile threats. https://www.elevenpaths.com/technology/tacyt/index.html
12. P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M.S. Gaur, M. Conti, M. Rajarajan, Android security: a survey of issues, malware penetration, and defenses. IEEE Commun Surv Tutor **17**(2), 998–1022
13. A.P. Felt, M. Finifter, E. Chin, S. Hanna, D. Wagner, A survey of mobile malware in the wild, in *Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices* (ACM, 2011), pp. 3–14
14. Y. Fratantonio, C. Qian, S. Chung, W. Lee, *Cloak and dagger: from two permissions to complete control of the UI feedback loop, in Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (, San Jose CA, 2017)
15. Google, AOSP source code for filesystem_config. https://android.googlesource.com/platform/system/core/+/master/libcutils/include/private/android_filesystem_config.h
16. M. Grace, Y. Zhou, Q. Zhang, S. Zou, X. Jiang, Riskranker: scalable and accurate zero-day android malware detection, in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (ACM, 2012), pp. 281–294

17. K. Nohl, J. Lell. *Mind the Gap—Uncovering the Android Patch Gap through Binary-only Patch Analysis* (2018)
18. E.J. Kartaltepe, J.A. Morales, S. Xu, R. Sandhu, Social network-based botnet command-and-control: emerging threats and countermeasures, in *International Conference on Applied Cryptography and Network Security* (Springer, 2010), pp. 511–528
19. Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, The new ext4 filesystem: current status and future plans. Proc. Linux Symp. **2**, 21–33 (2007)
20. K. Nohl, Mobile self-defense (snoopsnitch), in *Proceedings of Chaos Computer Security Conference* (2014)
21. Ed O'Keefe, https://www.washingtonpost.com/news/post-politics/wp/2013/06/06/transcript-dianne-feinstein-saxby-chambliss-explain-defend-nsa-phone-records-program/?utm_term=.f2e1466faae2 (2013)
22. C. Patsakis, E. Alepis, Knock-knock: the unbearable lightness of android notifications, in *Proceedings of the 4th International Conference on Information Systems Security and Privacy, ICISSP 2018, Funchal, Madeira-Portugal, January 22–24, 2018*, ed. by P. Mori, S. Furnell, O. Camp (SciTePress, 2018), pp. 52–61
23. N. Peiravian, X. Zhu, Machine learning for android malware detection using permission and API calls, in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI)* (IEEE, 2013), pp. 300–305
24. Android Police, https://www.androidpolice.com/2017/11/12/google-will-remove-play-store-apps-use-accessibility-services-anything-except-helping-disabled-users/ (2017)
25. C. Spensky, J. Stewart, A. Yerukhimovich, R. Shay, A. Trachtenberg, R. Housley, R.K. Cunningham, Sok: privacy on mobile devices–it's complicated. Proc. Privacy Enhanc. Technol. **2016**(3), 96–116 (2016)
26. Statista, Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 2nd quarter 2018. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/
27. T. Vidas, D. Votipka, N. Christin, All your droid are belong to us: a survey of current android attacks, in *Proceedings of the 5th USENIX Conference on Offensive Technologies* (USENIX Association, 2011), pp. 10–10
28. D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, K.-P. Wu, Droidmat: android malware detection through manifest and API calls tracing, in *2012 Seventh Asia Joint Conference on Information Security (Asia JCIS)* (IEEE, 2012), pp. 62–69
29. L. Ying, Y. Cheng, Y. Lu, Y. Gu, P. Su, D. Feng, Attacks and defence on android free floating windows, in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (ACM, 2016), pp 759–770
30. Y. Zhou, W. Zhi, W. Zhou, X. Jiang, Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. NDSS **25**, 50–52 (2012)