Understanding the Evolution of Android Applications

PH.D THESIS

Paolo Calciati
Understanding the Evolution of Android Applications

Submitted in partial fulfillment of the requirements for the degree of:
Doctor of Philosophy in Software and Systems

Author: Paolo Calciati
Advisor: Dr. Alessandra Gorla

September 2019

Thesis Committee:
Chair/Presidente: Dr. Juan Caballero, IMDEA Software Institute, Spain
Secretary/Secretario: Dr. Dario Fiore, IMDEA Software Institute, Spain
Member/Vocal: Dr. Federica Sarro, University College London, United Kingdom
Member/Vocal: Dr. Gregorio Robles Martínez, Universidad Rey Juan Carlos, Spain
Member/Vocal: Dr. Mariano Ceccato, Fondazione Bruno Kessler, Italy
Google released the first version of Android in September 2008, and it quickly grew to become the most popular mobile operating system, with a market share of over 75% as of May 2019. Not only the operating system has changed a lot in the last 10 years, but also applications have been constantly updated to keep up with both the continuous changes of the system and user requests. With such frequent updates, it can be difficult for final users or app store managers to understand what really changes between two releases of the same app, and whether updating an app could be a threat. In this thesis we focus on understanding how Android applications evolve, concentrating our research on analyzing subtle changes in the behavior that may pose security and privacy threats.

Our first contribution is an analysis on how permission requests evolve in Android applications. In this work we confirm some trends already reported in the literature, such as applications increasing the number of requested permissions over time, and also present new findings, such as showing that the addition or removal of permissions and functionalities is not always correlated.

Following the results of the first study, where we focused only on permission requests, we perform a broader and more in-depth analysis which takes in consideration multiple aspects of applications. We combine information extracted with different static and dynamic analyses, such as information flows, layout changes and libraries used, to report insights on the evolutionary behavior of applications. We show that new information flows in 18% of the cases incur without any change in the user interface, and thus are very hard for the user to notice.

The third contribution of this thesis is part of a collaboration on a study performed on parental control applications: those apps, which target a more sensitive user-base, should particularly pay attention to security and privacy issues. We show that these apps are on average more permission-hungry than the top 150 apps in the Google Play Store, and report some privacy issues discovered, such as the use of libraries which forbid in their Terms of Service their use in
children-oriented applications.

The fourth contribution is a study on the threat that permission groups in the Android permission model introduced with Android 6 can pose to users. With the new model dangerous permissions are divided into groups, and automatically granted by Android if an app already has another permission within the same group. We investigate how often developers ask for permissions that the OS would automatically grant, and if developers are taking advantage of this behavior to access sensitive user data, reporting that in fact this is the case.

Finally, all the analysis techniques that we implemented for our studies are integrated in Cartographer, an analysis framework created as part of the work of this thesis, and which we made available to the research community.
Google lanzó en septiembre de 2008 la primera versión de Android, que creció rápidamente hasta convertirse en el sistema operativo móvil más popular, con una cuota de mercado superior al 75% en mayo de 2019. No solo el sistema operativo ha cambiando significativamente en los últimos 10 años, sino también las aplicaciones se actualizan continuamente para estar al día con los constantes cambios del sistema y las necesidades de los usuarios. Con todos estos constantes cambios, puede ser difícil para los usuarios finales o los administradores de las tiendas de aplicaciones entender qué ha cambiado realmente entre dos versiones de la misma app, y si actualizarla puede ser una amenaza. En esta tesis nos centramos en analizar cómo las aplicaciones de Android evolucionan, centrándonos en analizar cómo las aplicaciones de Android evolucionan, centrándonos en la seguridad y la privacidad.

Nuestra primera contribución es un análisis de cómo las solicitudes de permisos evolucionan en aplicaciones de Android. En este trabajo confirmamos algunas tendencias ya reportadas en la literatura, como que las aplicaciones aumentan la cantidad de permisos solicitados a lo largo del tiempo. También presentamos nuevos hallazgos, como que la inclusión o eliminación de permisos y funcionalidades no está siempre correlacionada.

A continuación de los resultados del primer estudio, donde nos centramos solo en las solicitudes de permisos, realizamos un análisis más amplio y en mayor profundidad que tiene en cuenta varios aspectos de las aplicaciones. Combinamos los datos extraídos, como flujos de información, cambios en el layout y librerías usadas, para reportar el comportamiento evolutivo de las aplicaciones, mostrando que la inclusión de un flujo de información nuevo dentro de la aplicación es completamente transparente para los usuarios el 18% de los casos.

La tercera contribución en esta tesis es un estudio realizado sobre aplicaciones de control parental: estas aplicaciones, que tienen usuarios muy vulnerables, deben prestar especial atención a la seguridad y la privacidad. Mostramos que estas aplicaciones piden, de media, más permisos que las 150 aplicaciones más...
descargadas de Google Play Store. También reportamos algunos problemas de privacidad, como el uso de librerías que prohíben en sus términos de uso su inclusión en aplicaciones orientadas a su uso por niños.

La cuarta contribución es un estudio sobre la amenaza para los usuarios que los grupos de permisos introducidos en Android 6 pueden suponer. A partir de esta versión los permisos están divididos en grupos y si una app ya tiene concedido un permiso del grupo y pide uno nuevo, este se acepta de forma automática. Investigamos cómo de habitual es este comportamiento y si los desarrolladores están abusando de ello para acceder a información delicada de sus usuarios. Nuestros análisis demuestran que, efectivamente, esto se produce.

Finalmente, todas las técnicas de análisis implementadas para nuestros estudios fueron integradas en Cartographer, un framework de análisis creado como parte del trabajo de esta tesis y que hemos puesto a disposición de la comunidad investigadora.
To Mele
Thanks to my parents, Anna and Alessandro, for all the sacrifices you went through to make me the person I am today.

Thanks to my brothers, Andrea and Francesco, for the constructive discussions and the good advice.

Thanks to my sister, Eleonora, for the laughs, the good moments, and for making me feel like at home even when I am far away.

Thanks to Mele, for supporting me in all the important decisions and sharing with me the best experiences of my life.

Thanks to Chiara, for greeting me everyday with a smile.

Thanks to all my friends, old and new ones, for sharing the journey with me.

Thanks to Alessandra, for giving me the opportunity to pursue my PhD, and for being an advisor, a guide, and a friend.
1 Introduction 1
  1.1 The Android Ecosystem 3
  1.2 Permission Model in Android 4
    1.2.1 Recent Changes to the Permission Model 7
  1.3 Thesis Contributions 9
  1.4 Thesis Organization 10

2 Related Work 12
  2.1 Application Features 12
    2.1.1 Permission 13
    2.1.2 Third-party Libraries 14
    2.1.3 Dynamic Code Loading 15
    2.1.4 User Reviews 16
    2.1.5 Repositories and App Collections 16
    2.1.6 Malware Detection 17
    2.1.7 Automatic Input Generation 17
    2.1.8 App Privacy 18
  2.2 Android App Evolution 19

3 A Preliminary Study of Permission Request Evolution 21
  3.1 Dataset and Evaluation Process 21
  3.2 Empirical Results on Permissions Evolution 22
    3.2.1 Quantitative Study 23
    3.2.2 Qualitative Analysis 26
  3.3 Takeaways 28

4 Cartographer: Analyzing the Evolution of Android Apps 29
  4.1 Analyzing the Evolution of an App 30
4.1.1 App Releases Retrieval ........................................... 31
4.1.2 Application Metadata ............................................. 31
4.1.3 Library Analysis .................................................... 32
4.1.4 Apk Content Analysis ............................................... 33
4.1.5 Dynamic Analysis of Network Traffic .......................... 35
4.1.6 Static Analysis of Network Traffic with String Analysis .... 36
4.1.7 API Evolution ...................................................... 37
4.1.8 Dangerous Permission API Mapping ............................ 38
4.1.9 Data Leak Analysis .................................................. 38
4.1.10 Data Analysis ....................................................... 40

4.2 Empirical Study on Android Releases ............................ 42
4.2.1 Dataset .............................................................. 42
4.2.2 RQ1: How does a new Data Flow Correlate to Other Changes in the Release? ...................................... 44
4.2.3 RQ2: How do Web Domains Relate to Changes in the Layout? ................................................................. 45
4.2.4 RQ3: How do Information Flows Evolve During the Lifetime of an Application? ............................................ 46
4.2.5 RQ4: How do API Calls Evolve? ................................... 48

4.3 Parental Control Apps Analysis ..................................... 49
4.3.1 Dataset .............................................................. 49
4.3.2 Permission Analysis .................................................. 51
4.3.3 Third-party Library Analysis ...................................... 56

4.4 Limitations and Threats to Validity .................................. 60
4.5 Takeaways .............................................................. 61

5 Possible Threats of Automatically Granted Permissions .......... 62
5.1 Threat Scenario ......................................................... 63
5.1.1 Threats of Permission Groups ..................................... 64
5.1.2 Motivating Example .................................................. 65
5.2 Prevalence of Automatically Granted Permissions............... 66
5.2.1 Dataset Selection ...................................................... 67
5.2.2 Experiment Protocol .................................................. 68
5.2.3 Results .............................................................. 69
5.3 Actual Use of Automatically Granted Permissions ............... 70
5.3.1 Static Bytecode Analysis ............................................ 70
5.3.2 Results .............................................................. 73
5.4 Manual Analysis and Main Findings ................................ 74
5.4.1 Dynamic Analyses and Manual Inspection ..................... 74
5.4.2 Confirmed Data Leaks ............................................... 75
5.5 Limitations and Threats to Validity .................................. 79
5.6 Takeaways .............................................................. 79
6 Conclusions & Future Work

6.1 Future Research Directions

6.1.1 Automated Android Test Update

6.2 Conclusions

Bibliography
List of Figures

3.1 APK date distribution. ............................................. 22
3.2 Permission change. ............................................. 23
3.3 Permission changes vs. Android SDK version change. ........ 26
3.4 Permission evolution plot of the TabShop app. ............... 27
4.1 Main components and analysis flow of Cartographer. ........ 31
4.2 Libraries included in Smart Audiobook Player. ............... 34
4.3 Apk Structure of Mobile Print - PrinterShare. ................ 35
4.4 Saycupid Domains Heatmap. .................................... 36
4.5 Stringoid Heatmap example. ..................................... 37
4.6 TripAdvisor Flow. .............................................. 41
4.7 App Download Distribution. ..................................... 43
4.8 Yahoo Aviate Launcher Flow. .................................. 46
4.9 Noom Coach Flow. ............................................. 47
4.10 Security Master Flow. .......................................... 47
4.11 Data flows of PETATTO CALENDAR. ......................... 48
4.12 Permission heatmap. We list one app per row, indicating the permissions it requests. We differentiate dangerous permissions (blue), from normal (green), signature (orange), and custom (yellow). ................................................................. 52
4.13 Comparison of percentage of apps asking for dangerous permissions between Monitoring and Restriction apps. ............. 53
4.14 Percentage of apps using libraries belonging to the Advertisement (green), Analytics (orange) and Social Network (blue) categories. .......................................... 58
5.1 User interface of the BirthDayDroid app ....................... 65
5.2 Excerpt of the AndroidManifest.xml file of the modified BirthDayDroid app ............................................. 66
5.3  Android release distribution of our dataset (ApkPure and AndroZoo)  67
5.4  Automatically granted permissions in ApkPure  71
5.5  Automatically granted permissions in AndroZoo  72
5.6  Automatically granted permissions in AndroZoo and ApkPure combined  74
List of Tables

1.1 Android 6 dangerous permission groups ........................................ 6
1.2 Android 9 dangerous permission groups, We mark in the table the changes introduced in Android 8 (*) and (**) Android 9 ........... 8

3.1 Comparison between the patterns present in our dataset to the ones reported by Wei et al. [1] .............................................................. 24
3.2 Observed Patterns in permission evolution .................................... 24
3.3 Most frequently legitimate and overprivileged added permission ... 25

4.1 Potentially malicious domains ...................................................... 45
4.2 Most common flow patterns ....................................................... 46
4.3 Most common flow sources libraries .......................................... 48
4.4 Unusual permission requests. We report whether the apps have been unlisted from Google Play (GPlay column). ......................... 55
4.5 Classification of third-party libraries found in parental control apps. For each category we report the number of libraries (the total number of package names matching one of these libraries) and a short description of the category .......................................................... 57
4.6 Attribution of permission usage to code belonging to application or third-party libraries. For each dangerous permission we report whether it is used only in the app code, library code or in both. The last column reports uses that we cannot attribute for sure to app or libraries .................................................. 59
This thesis comprises three papers, the first two have been published to peer-reviewed academic conferences and journals. The last one is, at the moment of writing this document, under submission. The following list summarizes the aforementioned publications:

- Paolo Calciati, Alessandra Gorla
  **How Do Apps Evolve in Their Permission Requests? A Preliminary Study**

- Paolo Calciati, Konstantin Kuznetsov, Xue Bai, Alessandra Gorla
  **What Did Really Change with the New Release of the App?**
  In Proceedings of the 14th International Conference on Mining Software Repositories (MSR), May 2018, Gothenburg, Sweden [3].

- Paolo Calciati, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller
  **The Risk of App Updates with Automatically Granted Permissions**
  Under submission
In April 2017 Android has overtaken Microsoft Windows for the first time as the world’s most popular OS in terms of total Internet usage across desktop, laptop, tablet and mobile combined \(^1\). This growth means that the Android ecosystem is now a fast evolving, competitive market. To remain appealing for users and avoid them migrating to alternative apps, developers have to continuously update their apps to provide new features and address bug fixes as fast as possible. Frequently updated apps also have better visibility on the mobile stores. This fact, along with the high competition of the market, ensures that popular Android apps are released on a monthly or even weekly basis [4].

While frequent release cycles are beneficial to be ahead of the curve, they also cause problems for market managers and final users. Market managers, such as Google for its Google Play Store, need to analyze every single version before approving it for publishing to the store. Final users, on the other hand, receive updates for the apps installed on their devices transparently, as by default the Android system notifies them only when there are substantial changes in the list of permissions that the app requests. Informed users, however, cannot easily understand how the behavior of an app changes with an app release, as most of the changes happen beyond the user interface and the list of requested permissions, which is what users can easily analyze.

An app which already has access to the Internet may suddenly start establishing connections with suspicious hosts after a new release. If the app processes some sensitive data, such as the user’s contact list, it could suddenly start sending this information to a third party server. Without a proper understanding of what has changed in the new version of an app, it is hard to take an informative guess of whether an update is beneficial or harmful to the user’s security and privacy.

In the early releases of the Android OS, apps had to request all permissions statically, and users had to accept them all up front to install and use an application. To overcome this limitation, many researchers designed and imple-

\(^1\)http://gs.statcounter.com/press/android-overtakes-windows-for-first-time
mented security policy enforcement systems for Android framework following the
dynamic permission model, thus allowing users to block some of the requested
permissions [5]. Moreover, users originally had to approve every new app release
on their device if the update included any new permission request. Given the fre-
quency of updates and potential changes in the list of permission requests, most
app developers had the habit of asking for extra permissions in the first releases.
The rationale was that even if early releases were not using some of the requested
permissions, the user had to grant them anyway to use the app, and developers
could use these permissions in later releases without bothering the user for fur-
ther approval. Several researchers showed that this practice caused many apps
to be overprivileged and left space for potential security attacks [6]. Despite the
habit of including permissions that could be used in future releases, other studies
showed that there is anyway the tendency to ask for more permissions with new
releases of apps [1].

Time, user feedback and research evidence showed that the old permission
model was not user friendly, since most users were not checking or understanding
the list of permissions at all [7]. This led to a major change in the permission
model: first and foremost, as described in details in Section 1.2, starting from
Android 6 apps dynamically ask for permissions, i.e. they ask for user consent
when they use the permission for the first time. This change has the positive
effect that users can better understand when and how an app needs to access a
protected resource, and can decide to deny such permission. Secondly, to simplify
the permission model for user understanding, Android introduced permissions
groups.

With this change, users would be asked to grant a permission only if the
app asks for a permission that does not belong to any permission group that
was already granted before. This means that, for instance, if an app already
had the granted permission to access the coarse location of the user, it may
suddenly access the more precise location information –obtained through the
GPS– without further consent since accessing fine and coarse location belong
to the same permission group. Similarly, an app declaring the RECEIVE_SMS
permission, and thus capable of receiving SMS, could suddenly start sending SMS
without further user approval by adding the SEND_SMS permission, since these
two permissions belong to the same group. Changes in the list of permissions
are however worth notice, since they most likely represent a major change in the
behavior of the app.

In the following Sections we introduce the concepts necessary for the under-
standing of this work:

• Section 1.1 describes the Android ecosystem.
• Section 1.2 presents an overview of the Android permission model and its
evolution.
• Section 1.3 reports the main contributions of this thesis.

• Finally, Section 1.4 describes the organization of this document.

1.1 The Android Ecosystem

Android developers can distribute their apps through App Stores, such as the Google Play Store. App stores have been defined by Jansen and Bloemendal [8] as “An online curated marketplace that allows developers to sell and distribute their products to actors within one or more multi-sided software platform ecosystems”. Such stores allow users to easily access new app releases, as they can be downloaded by default on any mobile device.

Each application comes with a manifest file, which contains essential information about the app: the Java package name that uniquely identifies the app; its version name and version code; the list of Android components that the app offers—activities, services, broadcast receivers, and content providers; the list of permissions that the app requests to access protected parts of the API and interact with other applications; the minimum Android framework release required; and the list of native libraries that the app must be linked to.

Similarly to how the package name identifies an application, different Android releases of the same app can be distinguished thanks to two identifiers. The versionCode is a unique positive integer used for internal versioning, which can be used to determine whether one release is more recent than another. A greater value of the versionCode indicates that the release is successive to any other with a lower value. This number does not necessarily have a strong resemblance to the second identifier, the versionName, which specifies the release version visible to the user. VersionNames are not necessarily unique across releases and, for the Android system, this value has no purpose other than to be displayed to users. Most developers do however define versionName in the format major.minor.point, following semantic versioning, even though such convention is not enforced by the platform.

Google Play allows developers to publish multiple APKs for the same release to support devices with different requirements such as screen sizes and CPU architectures: those APKs share the same versionName.

Beside the manifest file, the APK contains all the binary code files of the app and resources. The content of an APK is divided into separate folders. Resources have their own folder. The drawable folder contains graphics to be shown on the screen, including bitmaps, shapes, and animations. The layout folder contains XML files that specify the user interface layouts. A layout defines the visual structure for a user interface, such as the UI for a screen or app widget.

---

The security model of the operating system is based on a permission mechanism, which we detail in Section 1.2.

Android applications are delivered as APK files: each of them is an archive containing bytecode of both application and external libraries, resources such as icons and images, and the AndroidManifest.xml. The manifest contains information about the application, specifying the package name, the developer, and among many other things the list of permissions declared by the application.

1.2 Permission Model in Android

Android protects the access to sensitive user data and critical system features by means of permissions: when an app wants to access such data or wants to use such features, it must request the corresponding permission first. Depending on the permission, the system might grant it automatically, such as when an app wants to connect to the Bluetooth, or might ask the user to explicitly approve it, as it happens when an app wants to access the phone’s GPS.

Android permissions fall into four categories, depending on the access they grant:\(^3\): normal, dangerous, signature and signatureOrSystem.

- **Normal** is the default permission value, giving requesting apps access to application-level features with low risk to the user, other applications or the system, such as Internet access. Those permissions are automatically granted by the system.

- **Dangerous** permissions give a requesting app access to private user data or control over the device and require explicit user agreement. If an app declares that it needs a dangerous permission, the user has to explicitly grant the permission to the app. Until the user approves the permission, your app cannot provide functionality that depends on that permission.

- **Signature** permissions are automatically granted by the system if the requesting application is signed with the same certificate as the application that declares the permission.

- **SignatureOrSystem** permissions, deprecated in API level 23, are used for special situations where multiple vendors have applications built into a system image and need to share specific features explicitly. They are automatically granted by the system if the apps are in a dedicated folder on the Android system image or signed with the same certificate as the application that declares the permission.

Finally, there are a couple of **Special** permissions, namely WRITE_SETTINGS and SYSTEM_ALERT_WINDOW, that are particularly critical and need to be

\(^3\)https://developer.android.com/guide/topics/manifest/permission-element
declared in the manifest and authorized by the user by having the app send an intent which opens a management screen.

In this thesis we focus primarily on dangerous permissions, although one of studies also considers other types of permissions, as some of them could harmfully impact the operating system or the user’s privacy [9].

The Android system asks the user to grant dangerous permissions in different ways, depending on the version of the system installed on the mobile device.

**Old Android permission model** If the device is running Android 5.1 or lower—corresponding to API level 22 or lower—or the app’s `targetSdkVersion` declared in the manifest is 22 or lower, the system asks the user to grant all dangerous permissions to the application at installation-time. If the user accepts, the system grants all requested permissions. By declining, instead, the user cannot use nor install the application. Moreover, the user then has to explicitly accept every new permission requested with following updates of the app. Without approval, the update process does not even start. Despite working only on quite obsolete versions of the Android system (i.e. pre Android 6.0), this permission model is still running on ∼25% of the Android devices as of August 2019 ⁴.

**Permission groups since Android 6.0** Android 6.0 introduced a big change in the permission model: if the device is running Android 6.0 (API level 23) or higher, and the app’s `targetSdkVersion` is 23 or higher, the system asks the user’s approval for each permission at runtime upon its first use. When this happens, Android shows a system dialog specifying which permission group the app wants to access, and asks the user to accept or deny it. A developer can also add a custom explanation message to help the user understand why the app needs that specific permission.

To help users making more meaningful and informed choices about granting or not a permission without overwhelming them with too many requests, the Android system organizes dangerous permissions related to similar functionalities into groups. For example the CALENDAR group contains both the `READ_CALENDAR` and the `WRITE_CALENDAR` permissions, as both permissions operate on the calendar. Table 1.1 shows the dangerous permissions and their relative groups in use in Android 6.0, which is the first release of Android using this permission model. As the Table shows, these groups have very broad functionalities.

When an app asks for a new dangerous permission, the system shows the permission request dialog only if the app did not request any permission in the same permission group before. Thus, if the app has already been granted another dangerous permission in the same group, the system grants that permission without

⁴https://developer.android.com/about/dashboards
Table 1.1: Android 6 dangerous permission groups

<table>
<thead>
<tr>
<th>Permission Group</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALENDAR</td>
<td>READCALENDAR, WRITECALENDAR</td>
</tr>
<tr>
<td>CAMERA</td>
<td>CAMERA</td>
</tr>
<tr>
<td>CONTACTS</td>
<td>READCONTACTS, WRITECONTACTS, GETACCOUNTS</td>
</tr>
<tr>
<td>LOCATION</td>
<td>ACCESSFINELOCATION, ACCESSCOARSELOCATION</td>
</tr>
<tr>
<td>MICROPHONE</td>
<td>RECORD_AUDIO</td>
</tr>
<tr>
<td>PHONE</td>
<td>READPHONESTATE, CALL_PHONE, READCALLLOG, WRITECALLLOG, ADDVOICEMAIL, USE_SIP, PROCESSOUTGOINGCALLS</td>
</tr>
<tr>
<td>SENSORS</td>
<td>BODY_SENSORS</td>
</tr>
<tr>
<td>SMS</td>
<td>SEND_SMS, RECEIVE_SMS, READ_SMS, RECEIVE_WAP_PUSH, RECEIVE_MMS</td>
</tr>
<tr>
<td>STORAGE</td>
<td>READEXTERNALSTORAGE, WRITEEXTERNALSTORAGE</td>
</tr>
</tbody>
</table>

any interaction with the user.

For instance, both RECEIVE_SMS and SEND_SMS permissions belong to the SMS group, as both permissions operate on SMSs. When an app requests a new dangerous permission, the system prompts the request dialog only if the app does not have any permission in the same permission group already granted. If an application already has the RECEIVE_SMS permission granted, and in a new release it requests the SEND_SMS permission, the latter permission would be automatically granted by the system.

This could allow a malicious developer to exploit the permissions already granted to an app to automatically gain access to sensitive user data without them even noticing it.

As of August 2019, Android 6.0 and 7.0 are the most prevalent releases among all Android devices, accounting to over 36% of the market share.
1.2.1 Recent Changes to the Permission Model

The update from Android 6 (API level 23) to Android 7 ⁵ (API level 24) did not have a strong impact on the permission model, with the addition of 4 signature level permissions.

With the release of Android 8 ⁶ (API level 26), Google partially addresses the concerns we raise in this thesis by introducing two new dangerous permissions to the PHONE group: ANSWER_PHONE_CALLS and READ_PHONE_NUMBERS. The two permissions are required to allow an application to answer incoming phone calls programmatically, via the `acceptRingingCall()` API, and to access the phone numbers stored in the device, respectively. Besides those two dangerous permissions, the update also introduces seven new normal and signature permissions.

Additionally, Android 8 introduces a number of changes on how some of the current permissions work. Performing a Wi-Fi scan requires either that the app has been granted either the CHANGE_WIFI_STATE permission, or a LOCATION permission. Applications can no longer get access to the user account unless either the authenticator owns the account or the user explicitly grants the access. Lastly, prior to Android 8, an app that requested a permission at runtime would also immediately be granted access to all other permissions in the same permission group. This allowed for example libraries to access data protected by permissions that were not requested. With the update, only the directly requested permission is granted, while the others in the same permission group are automatically granted the moment they are requested.

Android 9 ⁷ (API level 28) introduces further updates that protect the user’s personal data. First and foremost, there is a separate CALL_LOG permission group, and all the permissions related to call logs belong to this group. Separating these permissions from others related to the PHONE group reduces significantly the risk, for example, of an app with the READ_PHONE_STATE permission—and which can thus read the unique ID of the device—suddenly accessing the call logs of the phone. This recent change in the permission model might be another indicator that Google observed cases where developers abuse the previous permission model.

The update introduces a new dangerous permission in the PHONE group: the ACCEPT_HANDOVER. It is used to allow a calling app to continue a call which was started in another app, as well as two new normal permissions to receive NFC transactions events and allow apps to use biometric sensors.

A third change introduced in Android 9 aiming to protect the user’s privacy is to limit the access to sensors for background applications: apps running in background cannot access microphone or camera, and do not receive events from

⁵https://developer.android.com/about/versions/nougat/android-7.0-changes
⁶https://developer.android.com/about/versions/oreo/android-8.0-changes
⁷https://developer.android.com/about/versions/pie/android-9.0-changes-all
Chapter 1. Introduction

sensors using continuous, on-change or one-shot reporting modes, unless the app uses them on a foreground service, making them visible to the user. Another change introduced with Android 9 is that apps can no longer read phone numbers or phone state without having the READ_CALL_LOG permission, in addition to the other permissions that the app would normally require. Additionally, the READ_PHONE_STATE permission is required to access the device’s hardware serial number.

Moreover, similarly to what introduced in Android 8, to access SSID and BSSID values returned from the Wi-Fi scan `getConnectionInfo()` method and app must have both i) a LOCATION permission, ii) the ACCESS_WIFI_STATE permission and iii) location services enabled on the device. Table 1.2 shows the dangerous permissions and their relative groups in use in Android 9, summarizing the changes to existing ones introduced in Android 8 and 9 compared to Table 1.1.

Table 1.2: Android 9 dangerous permission groups, We mark in the table the changes introduced in Android 8 (*) and (**) Android 9

<table>
<thead>
<tr>
<th>Permission Group</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALENDAR</td>
<td>READ_CALENDAR, WRITE_CALENDAR</td>
</tr>
<tr>
<td>CALL_LOG(**)</td>
<td>READ_CALL_LOG, WRITE_CALL_LOG, PROCESS_OUTGOING_CALLS</td>
</tr>
<tr>
<td>CAMERA</td>
<td>CAMERA</td>
</tr>
<tr>
<td>CONTACTS</td>
<td>READ_CONTACTS, WRITE_CONTACTS, GET_ACCOUNTS</td>
</tr>
<tr>
<td>LOCATION</td>
<td>ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION</td>
</tr>
<tr>
<td>MICROPHONE</td>
<td>RECORD_AUDIO</td>
</tr>
<tr>
<td>PHONE</td>
<td>READ_PHONE_STATE, READ_PHONE_NUMBERS(<em>), CALL_PHONE, ANSWER_PHONE_CALLS(</em>), ADD_VOICEMAIL, USE_SIP, ACCEPT_HANDOVER(**)</td>
</tr>
<tr>
<td>SENSORS</td>
<td>BODY_SENSORS</td>
</tr>
<tr>
<td>SMS</td>
<td>SEND_SMS, RECEIVE_SMS, READ_SMS, RECEIVE_WAP_PUSH, RECEIVE_MMS</td>
</tr>
<tr>
<td>STORAGE</td>
<td>READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE</td>
</tr>
</tbody>
</table>
In October 2018 the Google Play store announced\(^8\) a new policy for applications which deal with sensitive data covered by the SMS and CALL\_LOG permission groups\(^9\). First, apps lacking default SMS, Phone, or Assistant handler capability may not declare use of those permissions in the manifest. Apps can only use those permission groups if they are actively registered as the default SMS, Phone, or Assistant handler, and shall stop using them when they’re no longer the default handler. Apps can only use the permission —and the data they can access through it— to provide critical core app functionality, and are not allowed to sell those data. Exceptions to the default handler requirement rule can be granted upon request, provided a critical feature of the app has no alternate viable methods, and that it does not provide potential privacy and security threats to its users. Apps that fail to meet policy requirements may be removed from Google Play. However, this restriction does not affect other third-party stores or the way Android OS runs apps.

1.3 Thesis Contributions

In this section we present the contributions of this thesis, following the chronological order to show the connections between the different chapters and how the thesis organically evolves over time.

**Evolution of permission requests analysis (empirical study).**

After defining the topic of the thesis, the first step of our work is to perform a preliminary study to understand how applications evolve in their permission requests. We start our study looking at permission requests because they are an easy-analyzable proxy of the underlying behavior of an application. The results of the study would confirm if applications become more demanding in terms of dangerous permissions over time, and whether it is necessary to use more thorough analyses to characterize behavior changes across releases that may affect the security and the privacy of the final user.

The result of the empirical study shows a clear trend of applications requesting more permissions over time, and shows that an application requesting a new permission or removing one from the manifest does not necessarily imply a change in the app’s functionalities.

**Cartographer - App Evolution Analysis Framework (artifact).**

The preliminary study confirms the potential of the research direction. While the literature contains many tools for analyzing and extracting data from Android apps, none of them is directed at studying the evolutionary aspect of the apps. We thus decide to model and build a framework with this goal in mind: we develop Cartographer, a framework which uses multiple third-party developed


\(^9\)https://play.google.com/about/privacy-security-deception/permissions/
tools, adapted to our needs, and combines their results to better understand the evolution of apps. The framework has two main components, one that extracts information from APKs, and another that performs an analysis on top of them. This latter part is customizable, and we implement a new analysis for each new study we perform. The final version of the framework implements all the analysis techniques which have been used for the empirical studies presented in this thesis. Cartographer can for example combine information flow extraction and layout changes, showing whether the user can see any visual difference in the app layout when a new information flow appears in the code.

**Analysis of applications with over 50 releases (empirical study).**

We use Cartographer to perform a study on the same dataset used in our former empirical study, comparing whether we are able to perform a more in-depth analysis which provides useful insights on the apps’ evolution. The results we obtain are quite interesting, showing for example that when an information flow leaking sensitive data from the app is added, in 18% of the cases it is completely hidden to the user, as layout and permission requests are unchanged.

**Analysis of Parental Control Applications (empirical study).**

We also use Cartographer for a specific analysis on parental control applications. This is part of a larger study, which also covers the analysis of Privacy Policies, Terms of Service, and a dynamic analysis of the applications to verify whether they share personal information, and in this thesis we report only the part related with the static analysis performed by the author of the thesis. In the first part of our analysis we focus on permission requests, pointing out the differences between restriction –filter contents or define usage rules– and monitoring –generate activity reports– applications. We follow the study by taking in consideration third-party libraries, manually classifying all libraries used in our dataset, checking whether any of them forbid their use for children applications in their Terms of Service, and investigating which companies are gathering data generated by analytics libraries.

**Automatically granted permission analysis (empirical study).**

In this study we analyze the threat that Android permission groups may pose to users: a dangerous permission is automatically granted by the system if, when requested, another permission is already granted in the same group. We analyze applications from two repositories, AndroZoo and ApkPure, to investigate how prevalent this practice is, how those permissions are actually used by the application, and report the cases we discovered where developers take advantage of automatically granted permissions to read sensitive data from the phone and send them over the Internet.

### 1.4 Thesis Organization

The reminder of this document is structured as follows.
Chapter 2 presents the related work, discussing the main contributions to the literature performed from other authors. We cite studies both concerning the analysis of Android applications features—such as API or user reviews—and the analysis of the evolution of Android applications.

Chapter 3 reports our introductory study on dangerous permissions, whose main objective is to analyze how applications evolve in their permission requests, focusing on dangerous permission requests. A second goal of this chapter is to familiarize ourselves with the subject by trying to confirm already reported trends in the literature, such as apps requesting permission which they do not later use in the code, or apps increasing the number of permissions requested over time.

Chapter 4 follows the previous study presenting Cartographer, a framework to extract and visualize evolutionary information of Android applications. With respect to the previous study, we perform a more in-depth analysis of applications, which still considers permissions, but also several other app features, such as libraries and information flows. Besides, we present the results of two empirical studies using Cartographer, the first conducted on 235 apps with at least 50 releases each, and the second conducted over 46 parental control applications.

Chapter 5 discusses the results of an empirical study on the threat posed by permission groups, introduced with Android 6.0, to the privacy and security of final users. We consider ~3 million Android apps, investigating how prevalent the practice of requesting permission which are automatically granted by the system is, how the data protected by those is being handled by the developers, and whether they are abusing the changes in the permission model to leak sensitive data over the Internet.

Finally, in Chapter 6 we present the conclusions of our work, where we summarize the most important findings and propose possible directions for future research.
In this chapter we present the related work, reporting the main contributions to the literature which focus on studying Android applications.

The chapter is divided into two parts. Section 2.1 presents the related work which targets single applications, either aiming at extracting information from the app, such as detecting specific behavior, or automatically generating input to exercise the app under analysis.

Section 2.2 focuses instead on research studying the evolution of Android applications, where multiple releases of the same app are taken in consideration to study the differences that arise between different versions.

### 2.1 Application Features

Android applications have been extensively analyzed, with researchers performing different studies on different aspects and features of the apps, ranging from detecting API compatibility issues [10, 11], to API usage [12, 13], to Android code smells [14, 15, 16], to Android bugs [17, 18, 19, 20].

The studies of Android apps are not only confined to the apps themselves, but also embrace external features, such as automatic input generation and app store analysis [21, 22].

In the reminder of this Section, we report a number of those studies, dividing them according to the focus of the research performed. However, we would first like to present a few examples belonging to different topics to give an insight of the extent of studies focusing on the Android environment.

Pauck et al. [23] and Qui et al. [24] both performed a comparison of taint analysis tools, as the comparing techniques used when publishing these tools are often not precisely described or differ in the leaks reported by the tools. Both studies emphasize the importance of providing detailed information when performing those comparisons.

The spread of mobile platform favored the growth of digital marketing and
advertisement, which in turn allows the creation of malware to inflate advertisement revenues. For instance, Facebook recently filed a lawsuit against two app developers accused of generating fraudulent revenue using the social media’s ad platform\(^1\).

This is an example of an advancement in technology that opens a new research direction, which later resulted in researchers focusing on the new topic. For instance, Dong et al. [25] provide a taxonomy of advertisement frauds and propose FraudDroid, a hybrid approach to detect advertisement frauds in mobile apps. The tool dynamically analyzes apps to build UI state transition graphs and collect the network traffic, checking them against a set of heuristics to identify whether ad frauds are being implemented.

Also concerning advertisement on Android apps, Gao et al. [26] performed a comprehensive analysis on advertisement schemes, analyzing three types of performance costs introduced by the use of ad: memory/CPU, traffic and battery. The authors observed that there there is a significant difference among the ad schemes taken in consideration, and that it should be taken in consideration when embedding ads into apps.

2.1.1 Permission

Due to the centrality of permissions in protecting the privacy of Android users, several research investigated their use by application developers [27, 28, 29, 30, 31, 32].

While permissions protect sensitive user data and certain system features, such as the camera, applications access those features by calling system APIs. As Google never released the API/permission mapping, several research has been done in this direction, with Au et al. [33] providing a mapping between permissions and their corresponding API calls.

Given the principal role of permission in protecting users’ security and privacy in Android applications, Felt et al. [34] conducted a user study to determine the perceived privacy and security risks of Android apps, and investigated how users should be confronted with permission requests, proposing a new approach which has been in part adopted when the Android permission model switched from install-time to runtime [35].

Felt et al. [6] performed the first study on permissions requested in the manifest of Android apps but not actually used in the application code, which are defined overprivileged permissions. The authors developed Stowaway, a tool to detect overprivileged in Android apps, applying it to a set of 940 apps, and reporting that one third of them are actually overprivileged. Out of those, many extra permissions could be attributed to developer confusion.

Felt et al. [36] investigate permission re-delegation: this occurs when an ap-

\(^1\)https://techcrunch.com/2019/08/06/facebook-clickjacking-ad-fraud-suit/
Chapter 2. Related Work

An application with a permission performs a privileged task for application without the permission. The authors report several vulnerabilities, which have later been confirmed as bugs.

Other works followed the same line of research, for example proposing an approach to test communications between apps [37]. Demissie et al. [38] investigated if malicious unprivileged apps can exploit the delegation model to exploit sensitive resources exposed by legitimate apps. The authors propose both a static and dynamic approach and concentrate on a specific vulnerability: an app executes privileged tasks on behalf of another app with data coming from the attacking app and without performing data validation.

Stevens et al. [39] analyzed permission usage and correlated it with Stack-Overflow questions regarding them, reporting that the likelihood of misusing a permission decreases with the popularity of the permission.

The research community also analyzed how app developers look for alternative ways to gain access to permission-protected user data. Reardon et al. [40] implement a test environment to instrument and analyze apps behavior and network traffic, uncovering a number of side and covert channels currently in use by hundreds of popular apps which allows them to gain unauthorized access to permission-protected sensitive user data.

Sadeghi et al. [41] build up on previous research [42, 43, 44, 45] of permission-induced security attacks —security breaches enabled by permission misuse— and propose a permission analysis and enforcement framework that considers the temporal aspects of permission-induced attacks for their detection and prevention.

The research performed by Sadeghi et al. [46, 47] also produced a number of artifacts for detecting permission-related security issues.

2.1.2 Third-party Libraries

Third-party libraries are a very important aspect of Android applications. Developers use them for several purposes, such as monetizing their app through advertisement, integrating new functionalities, or speeding up the development process.

An important amount of research focused on analyzing third-party libraries in Android applications [48, 49, 50, 51, 52], with a number of studies reporting that, as much as third-party libraries can be beneficial to the app development, some can exhibit questionable behavior, such as leaking private user information or tracking the mobile device.

For example, researchers at Palo Alto Networks discovered that Taomike SDK — one of the biggest mobile advertisement solutions in China— was uploading user’s SMS to an IP address belonging to the Taomike API server.

https://unit42.paloaltonetworks.com/chinese-taomike-monetization-library-steals-sms-messages/
Moreover, as Android applications compile both application and libraries code into the same output, it is a difficult task to identify which part of the code belongs to libraries, and which libraries the application is using. Various studies focused on creating tools to detect the third-party libraries used in an app.

Ma et al. [53] developed LibRadar, an obfuscation resilient tool that can detect third-party libraries in Android apps. To prevent code obfuscation from hindering the analysis, the authors resort the frequency of different Android API calls contained in the smali code, which are stable code features, and remain stable even when code is obfuscated. The signature of each smali folder is then generated by extracting the stable code features and representing them in a lightweight format using hashing.

On the next step, the authors use clustering on the package signatures to identify potential third-party libraries, analyzing over one million apps from the Google Play Store to gather a complete set of features for potential Android libraries. Given an input app, LibRadar extracts the folder signatures and compares them with the hashed features of all identified libraries, producing a list of possible libraries included in the input app.

Backes et al. [54] proposed LibScout, a library detection technique that is also resilient to common code obfuscation. The approach is similar to the one used by LibRadar, where a database with identified libraries is built, and the proposed tool extracts features resilient to code obfuscation techniques. In this case the extracted features are based on class hierarchy information, and allows the authors to determine the exact version of the library being used. Applying the techniques to the top applications of the Google Play Store revealed that developers often do not update third-party libraries to newer versions.

The updatability of third-party libraries was also studied by Derr et al. [55] which conducted a study over 1,2M apps, showing that 85.6% of the libraries could be updated without any code modification, and that 97.8% of libraries with a known security vulnerability could be fixed through a replacement of the library with the fixed version.

Zhang et al. [56] developed LibID, a library detection tool more resilient to code shrinking and package modification techniques. Similarly to the aforementioned studies, the tool builds library and application class profiles by relying on obfuscation resilient features, creating a textual representation of code features such as method calls or the existence of field-related operations. The extracted features for libraries are then compared against the ones of the application under analysis to determine which libraries it is using.

### 2.1.3 Dynamic Code Loading

Android allows developers to use specific API calls to dynamically load and execute code from external sources.

Some app stores examine applications before they are made available: the Google
Play Store, for example, implements a sourcing and vetting process which scans applications, aiming at securing the store against potentially harmful applications and applications not complying with Google’s policy. Developers could exploit dynamically loaded code to conceal the malicious behavior of their applications until they have been published, dynamically loading the malicious code at a later time, after the application has been installed on user devices. Several studies [57, 58, 59] analyze this issue more in detail, showing that several malware apps are indeed taking advantage of this feature to defeat the vetting process.

2.1.4 User Reviews

Application stores allow users to provide feedback to applications by means of posting an user review, which consists of a star rating and a comment. The reviews usually indicate the level of satisfaction with the application, but can also contain important pieces of information, such as bug reports or feature requests. Some literature focuses on analyzing users reviews, with a number of studies using sentiment analysis and topic analysis to extract and classify information from user reviews [60, 61, 62, 63, 64].

Due to the huge amount of user reviews generated, especially on popular applications, many researchers focused on tools and techniques to help summarizing information from different reviews and automatically identifying the most interesting ones. Fu et al. [65] developed a system to analyze user reviews which is able to detect inconsistencies between user comments, identify the main reasons users like an app, and learn how users’ complaints evolved over time. Panichella et al. [66] presented a taxonomy of review categories and proposed an approach which relies on natural language processing, text analysis and sentiment analysis to classify reviews into the different categories.

Nguyen et al. [67] published a study relating app reviews with security and privacy related changes. They showed that apps with a runtime permission handling receive a higher number of security and privacy reviews. The authors could also correlate almost half of the privacy-related reviews to third-party library code, showing that in most cases the reviews complained about a behavior introduced by third-party code.

2.1.5 Repositories and App Collections

Some researchers concentrated their efforts on producing apps repositories, made available to the research community to conduct further studies [68, 69, 70].

Allix et al. [71] released a dataset which currently contains over 8 millions applications crawled from several sources, both official, such as the Google Play Store, and third-party, such as Anzhi and AppChina. Each application is analyzed.

by different Antivirus products to identify potential Malware.

Krutz et al. [72] created a dataset of 4,416 Android releases, enhanced with information about applications’ development and maintenance, extracted using different static analysis tools.

In a similar way, Grano et al. [73] presented a dataset of 395 Android applications, including app reviews related to each version. They propose that the dataset could be used to study the evolution trends of quality metrics and code-smells.

2.1.6 Malware Detection

The research community has developed a number of ways to detect and investigate maliciously acting applications [70, 74]. We can divide the approaches used into three subsets: static, dynamic and hybrid.

Static analysis allows for the overestimation of the application code, often resulting in reporting false positives. This work mainly focuses on identifying an application as malicious by relying on studying some of the application features, such as by analyzing the permission requests [30, 75, 76, 77], the API calls [78, 79, 80, 81, 82, 83] or the application bytecode [84, 85, 86, 87, 88, 89].

Dynamic analysis aims at detecting malware by capturing the runtime execution of the app under analysis [90, 91, 92, 93]. Dynamic analysis will confirm the nature of all malware found, although it will fail to detect malicious behavior when it lies in a part of the application that is not explored during the analysis. For instance, TaintDroid [94] performs a taint analysis which tracks sources of sensitive data, such as accessing to the user account, and logs whenever one of those reaches a potential sink, such as an API call sending data over the Internet. Generally dynamic analysis relies on automated input generators to explore the application under analysis, such as Dynodroid [95] or DroidFuzzer [96].

Finally, some research combined static and dynamic analysis, creating hybrid approaches to get best of the two worlds [97, 98, 99, 100].

2.1.7 Automatic Input Generation

Due to the important growth mobile applications had in the past years, and in particular on the Android platform, many researchers investigated ways to automate apps testing. The literature contains multiple studies on techniques to automatically generate inputs for Android applications, including random exploration strategies [95, 101, 102], such as the Android’s application exerciser Monkey, model-based exploration strategies [103, 104, 105, 106, 107, 108, 109] or systematic exploration strategies [110, 111, 112].

Choudhary performed a thorough comparison of the main test input generations tools for Android [113], providing a clear picture of the state of the art and

---

4https://developer.android.com/studio/test/monkey
reporting that random exploration strategies, and in particular Google’s Monkey, achieved better coverage than more sophisticated strategies.

More recently, Adamo et al. [114] and Vuong et al. [115] presented reinforcement learning approaches to automate the GUI testing by using a test generation algorithm based on Q-learning to explore an app’s UI. Both studies report an increase in code coverage with respect to previous techniques.

2.1.8 App Privacy

In the last years some research focused on the privacy aspects of Android applications.

Ren et al. [116] perform a network traffic analysis on 7,665 versions of 512 apps to analyze what personally identifiable information appear in the traffic generated by those apps. The authors identify several trends such as slow https adoption, increase in collecting of personally identifiable information, and third parties being able to link user location and activity across different applications. The results of their analysis show that privacy has worsened over time and also that the information gathered by an application can change with different releases, limiting the validity of studies that focus on a single version of an app.

A similar study was performed by Razaghpanah et al. [117]: the authors focus on third-party services whose main function relies on collecting tracking information from users (advertising and tracking services). They analyze the network traffic generated by apps to understand where sensitive data ends up, also extending the analysis to both parent companies of the services, which could later combine and/or sell data to other companies, and how data flows across borders, with the corresponding impact of privacy regulations.

Ceccato et al. [118] conducted an observational study on users’ perception of the risks associated to installing apps. The study highlighted some discrepancies between how users perceive the privacy and security risks and how they actually behave.

The same team, with Demissie et al. [119], proposed an approach to identify anomalous sensitive information flows. Apps are grouped by topic according to their description, and then the sensitive information flows of apps under analysis are compared against the ones extracted from trusted apps grouped under the same topic, thus identifying anomalous flows that may violate user’s privacy. This work relies on clustering technique first proposed by Gorla et al. [120]. Al-Subaihin et al. [121] propose a framework to produce a improved, dynamic, automated and fine-grained categorization of apps based on the advertised functionality.
2.2 Android App Evolution

In this section we collect a number of studies that directly target the evolution of Android applications, analyzing different releases of each app and focusing on the differences that arise.

The literature contains multiple publications investigating the evolution of Android applications. Wei et al. [1] focused on third-party and pre-installed apps, analyzing the patterns that emerged in the evolution of permissions, and reporting a trend of apps becoming overprivileged and requesting more permissions over time. Taylor and Martinovic [122] performed a very broad study on over 1.6M applications: they took quarterly snapshots of the Google Play Store over a one year period and analyzed the evolution of dangerous permissions.

Wang et al. [123] performed a large-scale empirical study analyzing applications removed from the Google Play store between two snapshots taken in 2015 and 2017, finding out that nearly half of the apps presented in the first snapshots had been removed. The study focuses on understanding why applications are removed, and presents various interesting findings, such as that over 20% of removed apps are released by 1% of the developers.

Hecht et al. [124] presented PAPRIKA, a tool based on Soot and its Dexpler module [125] to monitor the evolution of mobile apps quality based on general object-oriented and Android-specific anti-patterns. They analyzed a dataset of 106 applications with 3,568 releases, but were unable to identify general evolution trends.

Zhang et al. [126] examined the applicability of Lehman’s laws of software evolution on mobile apps, performing a case study on two applications. They focused on three laws, finding similar trends between mobile and desktop apps for two of them, while they could not conclude whether the third one holds true.

Book et al. [50] investigated Android ad libraries’ change in behavior over time by investigating 114,000 apps, extracting ad libraries and checking which permissions they try to access, based on the APIs they invoke. In their study the authors found out that the use of most permissions increased, and that more libraries use permissions that can pose privacy and security issues to users.

Martin et al. [127] used Causal Impact Analysis to identify causally significant releases in app stores. The authors developed a tool to analyze over 26K releases and report that overall release frequency seems not to be correlated with app success, but that there is evidence that price release description can impact the app’s success.

Nayebi et al. [128] performed an analysis of over 200,000 commits from the evolution of 1,519 open source Android apps, focusing on the deletion of functionalities. In their study the authors investigate which functionalities were deleted, analyzing how and why. According to the developers’ survey, only 5% of deleted functionality is related to the security improvement, whereas more than 50% of developers would consider to remove some functionality in response to the nega-
tive user reviews.

He et al. [129] performed an empirical study on evolution-induced compatibility issues in Android applications: consecutive Android releases can bring a very high number of changes, with 140 types, 1,500 methods and 979 fields introduced on average in each of the 11 new Android releases taken in consideration. The study reports that three quarters of newly added methods introduced with a new Android release are not supported by the Android Support library, and that over 90% of the apps need to address compatibility issues introduced by new Android releases.
In this chapter we present a preliminary study to understand how apps evolve in their permission requests across different releases. We analyze over 14K releases of 227 Android apps, and we see how permission requests change and how they are used. We find that apps tend to request more permissions in their evolution, and many of the newly requested permissions are initially overprivileged. Our qualitative analysis, however, shows that the results that popular tools report on overprivileged apps may be biased by incomplete information or by other factors. Finally, we observe that when apps no longer request a permission, it does not necessarily imply that the new release offers less in terms of functionalities.

The Chapter is structured as follows: Section 3.1 presents the evaluation process of the empirical study dataset we used to conduct it. Section 3.2 reports the empirical results on the analysis performed, discussing the results of both the quantitative and qualitative studies. Section 3.3 summarizes the findings and results of our study.

### 3.1 Dataset and Evaluation Process

The first step of our preliminary empirical evaluation is to obtain a dataset containing multiple releases of each app we want to test. For this task we resort to Androzoo [71], an online repository which comprises a collection of over 5 million Android apps, retrieved from different sources. Since this study does not focus on malware, we select the Google Play store as the only source to consider, given that several studies have shown that the apps distributed through this channel are largely trustworthy [130, 131, 132]. From this selection, we consider only apps with at least 50 releases, to have enough data for a study on apps evolution. The resulting set contains 14,880 releases of 235 different applications.

We analyze each release with Androguard, a tool that can extract declared
permissions and report the overprivileged ones. The Android platform allows developers to declare new, application-specific permissions, however in this study we focus only on Android official permissions, thus considered the ones matching the format \texttt{`android.permission.XXX'}, and leaving out app-specific permissions. Androguard crashes while analyzing the Dalvik bytecode of several releases of 10 different applications. We decided to keep in our analysis only the apps for which Androguard could analyze at least 50 releases: with this additional pruning, our final dataset comprises 227 applications with a total of 14,450 releases.

The distribution of releases per app goes from a minimum of 50 up to 171, with a mean value of 64. The APKs in the dataset have been released within the timeframe of August 2008 – January 2017, with the distribution shown in Figure 3.1. Out of the APKs in our dataset, 727 (5.03\%) do not have a valid release date, as reported by the Androzoo developers, and are represented in the plot as n/a.

To further understand the nature of the apps in our dataset, we look at the category, number of downloads, and rating as reported in the Google Play store. We conclude that the dataset is quite varied, since it covers all the 32 Android categories, but it focuses mainly on high quality and popular applications, with an average of over 44M downloads per app, and an average star rating of 4.29 out of 5. All the applications in our dataset are still active and available on the Google Play store, except for 8, which either changed package name or have been discontinued.

### 3.2 Empirical Results on Permissions Evolution

At first we perform a quantitative study to evaluate how permission requests evolve over time. We later conduct a qualitative study to have a more in depth

---

1. https://github.com/androguard/androguard
2. https://androzoo.uni.lu/lists
understanding of the phenomena observed in the first study.

3.2.1 Quantitative Study

The goal of the quantitative study is to look at how permission requests change across the lifetime of an APK. More precisely, we look at the following research questions:

- **RQ1: How do permission requests evolve?** We analyze how the permissions list changes between a release and its following one. We also look for specific patterns in the evolution, such as whether a permission is first requested and actually used only later.

- **RQ2: Which are the most frequently added permissions?** We check which permissions developers tend to add over time, and whether they are legitimate or overprivileged. Given the new group permission model introduced in Android 6 (explained in Section 1.2) we also check which are the most commonly added permissions that would not need any further approval of the user.

- **RQ3: Do permission changes concentrate within specific time periods, and in particular correlate with changes in the Android SDK?** In this final study we try to understand if there are relevant facts related to permission changes.

**RQ1: How do permission requests evolve?**

We extract the manifest and see how the list of declared permission changes between two consecutive releases. We observe that for 13,637 out of 14,223 release transitions, which accounts to 95.88% of the total number, there are no changes in the list of requested permissions.

Figure 3.2 shows the distribution of added and removed permissions on each version change after taking the versions without any change out of the equation.
Table 3.1: Comparison between the patterns present in our dataset to the ones reported by Wei et al. [1].

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Our results (%)</th>
<th>Results in Wei et al. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 → 1</td>
<td>80.18</td>
<td>90.46</td>
</tr>
<tr>
<td>1 → 0</td>
<td>15.27</td>
<td>8.59</td>
</tr>
<tr>
<td>1 → 0 → 1</td>
<td>3.27</td>
<td>0.84</td>
</tr>
<tr>
<td>1 → 0 → 1 → 0</td>
<td>1.27</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 3.2: Observed Patterns in permission evolution.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Count</th>
<th>Frequency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 → 2</td>
<td>219</td>
<td>24.39</td>
</tr>
<tr>
<td>0 → 1</td>
<td>141</td>
<td>15.70</td>
</tr>
<tr>
<td>2 → 1</td>
<td>90</td>
<td>10.02</td>
</tr>
<tr>
<td>0 → 1 → 0</td>
<td>50</td>
<td>5.57</td>
</tr>
<tr>
<td>1 → 0</td>
<td>48</td>
<td>5.35</td>
</tr>
</tbody>
</table>

On average we observe 0.64 permission additions per release, showing a clear trend of apps increasing the number of asked permissions over time.

As depicted in the plot, most changes involve only adding a single permission (287 occurrences). We observe only 78 occurrences of adding two permissions at the same time, while only 42 involving three or more permissions added. Permissions are rarely removed (151 occurrences in total), and most of the times this involves a single permission.

Similarly to what Wei et al. did in their work [1], we look for specific patterns in our dataset: Table 3.1 compares results obtained by Wei et al. with ours. We keep the same notation, where ‘0’ represents a non requested permission, ‘1’ represents a requested permission, and ‘→’ represents a state transition. Considering the fact that we have a much larger set of releases for each application (more than 50 versus an average of less than 4), and thus it is more likely for us to find longer patterns, the distribution found in [1] finds confirmation in our results despite the slight difference in the reported percentages. The patterns we consider in this comparison account for 83% of patterns found in our study, with the remaining 17% containing the 0→1→0 pattern, and longer patterns in negligible percentages.

Our analysis extends to cover also overprivileged permissions. For the rest of the section, we refer with ‘1’ to a legitimate permission, and with ‘2’ to an overprivileged one: Table 3.2 shows the most frequent patterns we discovered. Surprisingly, the most frequent pattern is to add a permission that remains overprivileged for the whole lifetime of the app. We further analyze these trends in Section 3.2.2.
### RQ2: Which are the most frequently added permissions?

Given that the previous research question showed that the most common trend is to add permissions, whether legitimate or overprivileged, we look at which ones are most frequently added. Table 3.3 shows the most frequent among the 646 permission addition found.

Taking into account the permission model of Android 6, explained in Section 1.2, we further analyze each newly added permission, and verify if it belongs to a permission group that was already granted in the previous release: in 35.75% of the cases (231 permission addition out of 646 in total) the newly added permission indeed falls in the same group of one of the already granted permission.

READ_EXTERNAL_STORAGE is the permission for which this behavior happened the most: many apps add this permission while they already declare WRITE_EXTERNAL_STORAGE. What is apparently unclear to developers is that apps granted with WRITE_EXTERNAL_STORAGE implicitly have the read access to the external storage as well, so there is no need to additionally ask for the READ_EXTERNAL_STORAGE permission.

A different story regards accessing the user’s location through either the ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION permissions, which belong to the same permission group but have information from different sources (from the network location—such as cell towers and Wi-Fi—and from the GPS, respectively). We observe that many apps initially use just coarse location, and request the ACCESS_FINE_LOCATION afterwards. Thus, without any further user consent or notification, they can obtain more precise location information.

<table>
<thead>
<tr>
<th>Permission (legitimate)</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_COARSE_LOCATION</td>
<td>58 (19.33%)</td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>49 (16.33%)</td>
</tr>
<tr>
<td>ACCESS_FINE_LOCATION</td>
<td>49 (16.33%)</td>
</tr>
<tr>
<td>CAMERA</td>
<td>32 (10.67%)</td>
</tr>
<tr>
<td>READ_CONTACTS</td>
<td>31 (10.33%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Permission (overprivileged)</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_EXTERNAL_STORAGE</td>
<td>112 (32.37%)</td>
</tr>
<tr>
<td>WRITE_EXTERNAL_STORAGE</td>
<td>25 (7.23%)</td>
</tr>
<tr>
<td>READ_SMS</td>
<td>25 (7.23%)</td>
</tr>
<tr>
<td>CAMERA</td>
<td>25 (7.23%)</td>
</tr>
<tr>
<td>RECEIVE_SMS</td>
<td>24 (6.94%)</td>
</tr>
</tbody>
</table>
Chapter 3. A Preliminary Study of Permission Request Evolution

RQ3: Do permission changes concentrate in specific time periods?

The last step of our quantitative analysis looks at the distribution of releases over time. More precisely we want to see if releases are more frequent in specific time frames. We thus divide the time frame in year quarters (x axis in Figure 3.3), and we report the percentage of permission changes that we observe in that quarter (blue bars in the plot). We can observe that there are peaks in the number of releases in q3 2013 and q3 2015, in our opinion due to the low amount of releases in the preceding quarters.

To further understand this trend, we list changes occurring in two subsequent releases that involve either the target or the minimum Android SDK. We plot, as yellow bars, the percentage distribution of such changes on the same time frame in Figure 3.3. We see that the two distributions have similar trends, suggesting that a change in the minimum or target SDK might trigger permission changes. We checked whether high change peaks would correspond to release dates of a new Android SDK, but we could not find a positive correlation.

3.2.2 Qualitative Analysis

The results of the quantitative analysis reported in Section 3.2.1 show that several apps have interesting patterns in their permission requests. Some apps, for instance, add and remove the same permission multiple times in their lifetime, as if some functionalities were added and removed from the app. In many other cases, instead, developers seem to ask for permissions they never use. To further analyze the permission evolution of the apps in our dataset, we plot legitimate and overprivileged permission requests as displayed in Figure 3.4. On the y-axis we report the permissions that the app requires at least once in its lifetime, and on the x-axis we report all the releases for that app. We use white, light blue and dark blue to represent not requested, legitimate and overprivileged permissions,
respectively.

Figure 3.4 presents the permission evolution of TabShop, a free shop keeping, cashier and point of sale (POS) app, which presents a quite interesting permission evolution pattern. Among the functionalities it offers, there is the possibility to scan barcodes. CAMERA thus seems an essential permission for the whole lifetime of this app.

The plot, instead, shows that the CAMERA permission presents a peculiar pattern: it is initially requested but not used up to release 60, it is later legitimately used for several releases, then it appears as overprivileged for a few releases before finally disappearing in the latest releases analyzed. We installed and ran different APKs of this app in the emulator, and found out that in the first releases it uses an intent to launch “Barcode Scanner”, an alternative app to scan barcodes. Between release 60 and 110 the app instead implements the feature itself, thus using the device’s camera directly. Within this timeframe (i.e., between release 60 and 110) the CAMERA permission appears twice in the manifest. From version 114 to version 117 the app switches back to using the intent to “Barcode Scanner”, but removes only one CAMERA permission from manifest, thus leaving the duplicate and making it overprivileged. Developers finally removed the unnecessary permission in version 118.

The qualitative analysis on this app gives a clear example of how adding or removing a permission does not necessarily imply the implementation or removal of the corresponding functionality, as it could rather be implemented in an alternative way.

This app is not the only one in our dataset that declares duplicate permissions. More precisely, 52 apps of the 227 we analyzed show this problem in at least one of their releases, with Firefox Beta topping the list with duplicated permissions in over 100 releases. Similarly, we observe that many apps include misspelled permissions (e.g. VIBRATION instead of VIBRATE). Although these cases are harmless, since the app asks for a permission that either does not exist or it already uses, they represent a bad practice, and may falsely report alarms.

Moreover, we notice that Androguard incorrectly reports other permissions as overprivileged. This happens for example for WRITE_EXTERNAL_STORAGE, which is also one of the most frequently reported permissions as overprivileged. To verify our claim we install several apps on the Android emulator removing the permissions reported as overprivileged, and we expect the application to crash if
the permission were to be used in the app code. We test for example version 9 of `com.popularapp.periodcalendar`, released in February 2013, removing WRITE and READ EXTERNAL STORAGE permissions, which were both labeled as overprivileged. Running the repackaged app leads to a crash when it tries to read or write some information on the SD card, showing that the WRITE.EXTERNAL STORAGE is legitimate, in spite of what was reported by Androguard.

The incorrect information reported by Androguard may be due to different causes: 1) it is known that a complete and correct mapping between API calls and permissions does not exist yet [33, 133, 6]; 2) I/O operations may be implemented in native code, preventing Androguard from seeing them; 3) finally, we also identified a bug in Androguard, which causes the analysis to always use the API mappings relative to API 19, even when the app targets another release. Thus, the lesson we learned is that trusting tools such as Androguard blindly may bias significantly the results.

### 3.3 Takeaways

The aim of this study is twofold: on one side we want to continue the analysis of the evolution of Android applications by providing a more in-depth study with respect to what had already been proposed in the literature. On the other side we want to familiarize ourselves with the Android environment, choosing an accessible entry point –the analysis of permission requests– into the subject.

Our findings confirm some issues with the Android permission model already reported in the literature, confirming the trend of of apps increasing the number of requested permissions over time. We also show some confusion between developers regarding the use of permissions, as suggested by our finding of mislabeled, duplicated and overprivileged permissions. Finally, we show that looking at permission requests, despite being able to give an hint of what an application does, is not a reliable way to identify all of its functionalities.
The mobile app market is evolving at a very fast pace. In order to stay in the market and fulfill user’s growing demands, developers have to continuously update their apps either to fix issues or to add new features. Users and market managers may have a hard time understanding what really changed in a new release though, and therefore may not make an informative guess of whether updating the app is recommendable, or whether it may pose new security and privacy threats for the user.

We propose a ready-to-use framework to analyze the evolution of Android apps. Our framework extracts and visualizes various information—such as how an app uses sensitive data, which third-party libraries it relies on, which URLs it connects to, etc.—and combines it to create a comprehensive report on how the app evolved.

Besides, we present the results of two empirical studies. The first is conducted on 235 applications with at least 50 releases using our framework. Our analysis reveals that Android apps tend to have more leaks of sensitive data over time, and that the majority of API calls relative to dangerous permissions are added to the code in releases posterior to the one where the corresponding permission was requested.

The second study is conducted over 46 parental control applications with 419 releases, and highlights security and privacy threats applications targeting children. The study reveals that those apps are more permission-hungry with respect to the average top application of the Google Play Store, and that some of them are using libraries which forbid their use for children in the Terms of Service.

The Chapter is structured as follows: Section 4.1 describes Cartographer, the framework we developed to analyze the evolution of Android apps. Section 4.2 presents the results of the empirical study on the evolution of Android apps. Section 4.3 reports the results of the empirical study on parental control applications. Section 4.4 discusses the limitations of our work. Section 4.5 summarizes
the findings and main takeaways of the two empirical studies.

4.1 Analyzing the Evolution of an App

There are many features that one might look at in order to understand how the behavior changed with a new release of the app. At a very high level, one may look at the list of requested permissions and assume that no major change occurs in the functionality of the app unless this list changes. Still, many subtle things can happen without any change in the list of permissions. For instance, an app that already had access to sensitive data, such as the contact list, may suddenly send this information to a third party server.

Our previous study showed that the vast majority of apps do not change their permission list across many releases. As a consequence, to properly understand the behavior of a specific release we have to resort to more comprehensive analyses.

Our framework aims to thoroughly analyze the behavior of an app keeping into account several aspects, each of them requiring a specific feature: we analyze the network traffic to have an understanding of which hosts the app talks to; we analyze how sensitive data flows across the application, and most importantly if there is any leak through the Internet; we analyze the evolution of API calls relative to dangerous permissions to discover how developers access sensitive information during the app’s lifecycle. We then visualize the output of each analysis to make it easy for the users to spot any significant difference.

Figure 4.1 shows the workflow of Cartographer and its main components. The workflow is divided into three logical parts. The first one aims to retrieve a significant amount of releases for the app of interest. The second one is to analyze each APK in isolation using different analyses. The third and the last one aggregates data and visualizes them for the user.

Each module in the information extraction workflow is implemented within the Calappa toolchain [134] on top of Luigi \(^1\). Luigi is a Python library that helps developers build complex pipelines of batch jobs. Each module can be executed separately, though, they are dependent on each other. To exchange information, modules produce JSON files. To achieve scalability, there is an ssh adapter that allows to execute tasks on several machines.

Cartographer visualizes relevant information in the form of a heatmap. As a general pattern, for each analysis, the heatmap shows on the x-axis the release version numbers and on the y-axis the features extracted from the corresponding analysis. We also complement these data with the location where the feature was found—in case of a third-party library we append its name separated by a colon. Columns can be highlighted with a colored flag: red if the version name of that release is the same as the previous one, dark green if the version name has

\(^1\)https://github.com/spotify/luigi
a major change, and light green for minor updates, thus helping to understand whether changes are related to the release cycle.

We now proceed to describe each analysis that Cartographer employs and its corresponding module.

### 4.1.1 App Releases Retrieval

To understand how the behavior of an app changed, it is first of all necessary to retrieve as many previous releases as possible. Cartographer supports different kinds of Android app sources. It requires the package name of an app or a list of packages. We implement adapters for Androzoo [71] and for the F-Droid app store \(^2\) to pre-filter and download app releases that satisfy certain filtering conditions. After the downloading process is complete, Cartographer creates an index list of APKs, which is fed as input to the Luigi modules to start the analyses.

### 4.1.2 Application Metadata

As described in Section 1.1, APK files contain essential information about the app stored in the Android manifest file. We extract basic information that is fed to other modules to enrich their analysis.

---

\(^2\)https://f-droid.org
We use the Android Asset Packaging Tool (aapt) \(^3\) to extract the following data:

- package name
- version code
- version name
- platform build version name
- available activities
- Android API level
- requested dangerous permissions

First, we verify that package name and version code are consistent with what the APK file name states and report possible mismatches. Next, we heuristically identify the release type, looking at the version name. Usually, this release is a string in the format `major.minor.point`. A change in the first part of the version name may indicate a major release update, which is expected to have substantial differences. A change in the second part may identify a release with small changes introduced, while a change in the last part should imply just minor changes and some bug fixes.

Google Play allows to publish different APKs each targeting different Android devices: these releases will have the same version name. If Cartographer finds two consecutive versions with the same version name it marks them with a red flag on the heatmap, because differences in these releases could, and should, be due to different architectures and not to radical changes in the app’s evolution.

We rely on Androguard \(^4\) to extract the list of overprivileged permissions. We define them as permissions that are declared in the AndroidManifest.xml file, but for which we could not find any corresponding API use in the application code.

Finally, we use the APK Parse library \(^5\) to extract all the activities listed in the Android Manifest, which we later use to compute the activity coverage of dynamic analysis.

The output of this module is a JSON file for each application package, containing all the aforementioned information for all the relative APKs in our dataset.

### 4.1.3 Library Analysis

Android developers resort to third-party libraries to provide many ready-to-use functionalities and services. They may also include third-party libraries that show ads only to get some revenue. When analyzing the behavior on an app, it is important to distinguish whether it comes from the core of the app (i.e. the

---

\(^3\) [developer.android.com/guide/topics/manifest/uses-feature-element.html](https://developer.android.com/guide/topics/manifest/uses-feature-element.html)

\(^4\) [https://github.com/androguard/androguard](https://github.com/androguard/androguard)

\(^5\) [https://github.com/tdoly/apk_parse](https://github.com/tdoly/apk_parse)
code that the app developer wrote) or from a third-party library, and thus even developers may not be fully aware of it.

The binary code, comprising an app and libraries, comes as a single package, and thus the identification of libraries in Android is not straightforward, especially when code is obfuscated, as proven by many studies performed on the subject [53, 135, 54]. In order to detect third-party libraries used in Android apps, we resort to LibRadar [53]. It detects libraries based on stable API features, which are obfuscation resilient. As LibRadar relies on known patterns to identify libraries, it can only recognize the ones already presented in its model.

Using our own heuristic we complement the information extracted by LibRadar to increase the number of identified libraries. We first use Soot [136] to collect the package names from the full names of classes inside an APK. For each package name we extract the prefix by removing package components until either the length of the prefix is less than 8 characters or there are only two components remaining. For example, ak.alizandro.smartaudiobookplayer would be reduced to ak.alizandro. We finally remove all the package names whose prefix matches the application package name prefix and identify the remaining packages as libraries. In this respect, we assume that developers keep the same package prefix across apps when reusing code.

We also truncate packages that contain obfuscated suffixes (e.g. com.paypal.a.b to com.paypal), and map completely obfuscated packages (e.g. com.b.d or a.b.c) to the specific OBFUSCATED tag.

To improve readability we manually created a mapping of known package names to the corresponding libraries (such as com.paypal to Paypal). We leave the package name as is for less known libraries. This simple heuristic does not work for obfuscated packages but still allows to noticeably enhance LibRadar’s results.

For each application, we produce a heatmap containing on the y-axis the libraries that the app includes, highlighting which ones are detected by LibRadar (dark blue cells), and which ones are added by us (light blue cells). Figure 4.2 is an example of such heatmap for the Smart Audiobook Player app. Some obfuscated libraries, as well as the Android Support v4 library are used by the app throughout the whole examined lifecycle. During the app’s lifecycle developers started adding more libraries, such as the Unseen BASS library, which provides sample, stream, and recording functions.

### 4.1.4 Apk Content Analysis

Cartographer extracts the resources contained in the APKs, aiming to understand if there are major changes with respect to previous releases. The focus is mostly on the UI, which is the principal channel through which the user perceives changes
in the application. We resort to Apktool \textsuperscript{6} to extract resource files, contained in the \texttt{res} folder, including strings, drawable resources, and application layout files. In this step we also extract the application code files, contained in the \texttt{smali} folder.

We use this information to show which changes happen between releases at the code and layout levels \cite{137, 138}, generating heatmaps as output. We compare two releases by analyzing the content of each file present in both packages. If a file is available in only one of the two APKs, we assume that it was either added or removed.

The resource analysis module outputs two heatmaps: the first one shows relevant information extracted from the Android manifest (i.e. how many activities, permissions, providers, receivers and services have been added, removed or changed with respect to the previous version). The second one, shown in Figure 4.3, reports the changes in code and UI elements between consecutive releases of the app.

The heatmap can be read as follows:

- the first two rows show how many UI elements are added or removed. This measure tracks changes visible to the user, such as adding or removing a button from an Activity.

- the next three rows (layouts added, removed and changed) show the percentage of layout files that have been added, removed or changed, giving a rough idea of how much the layout has been changed from the previous version.

- the following three lines (layout modification, insertion and deletion) report

\textsuperscript{6}\url{https://ibotpeaches.github.io/Apktool/}
the percentage of lines in the layout files that have been added, removed or modified.

- the drawable and values rows show the percentage of files changed, added and deleted in the `drawable` and `values` directories, which are found inside the `res` folder and contain application’s images for different screen sizes and localized strings.

- finally, the smali row shows the percentage of the code that has been added, deleted or modified (file based).

The heatmap cells show the percentage of changes with respect to the previous release. We can see from this heatmap that some versions, such as 122 and 123, have no changes in the layout at all, while in versions 131 and 138 developers heavily modified the layout. Unfortunately, this comparison is not applicable in case of obfuscation.

### 4.1.5 Dynamic Analysis of Network Traffic

With this module we want to monitor the network traffic generated by the application to discover potential security and privacy threats. The dynamic analysis uses Monkey, an automated event generator created by Android developers, to
generate pseudo-random streams of user events. Despite the high number of tools available for automatically exercising the application, Monkey was proved to be one of the most effective [113]. We use Monkey to execute each APK, with three runs of 5 minutes each, while logging the data produced by the tcpdump command.

We use the Bro Network Security Monitor ⁷ to extract the domains to which each APK connects and then analyze each of them on VirusTotal ⁸ to verify if they are malicious or not. We consider a domain as potentially malicious if it is reported as such by at least three VirusTotal detection engines.

For each application we also compute the percentage of activities covered, extracting all visited activities with the logcat command on the emulator and comparing them to the total number of activities extracted by the module that analyzes the manifest file. With our approach we covered, on average, about 20% of the activities.

Figure 4.4: Saycupid Domains Heatmap.

Figure 4.4 shows the output of the dynamic analysis for the Saycupid application: in the heatmap we highlight in blue which releases connect to which domains. From this example we can see that, from version 355 on, the application starts connecting to some third party analytics services (crashlitics.com and app-measurement.com).

4.1.6 Static Analysis of Network Traffic with String Analysis

As dynamic analysis might miss relevant parts of the code, we also implement a module that inspects the bytecode for network activity. This module relies on

⁷https://www.bro.org/
⁸https://www.virustotal.com
Stringoid \[139\], a static analysis tool that takes as input an APK and produces a set of string patterns representing URLs. We use it to extract constructed URL strings from applications, estimating the domains the app connects to.

For each extracted URL, we use the data obtained by Apktool and LibRadar to understand if it is used in the main application code or in a library, and in the latter case we identify in which one. The output of the Stringoid task is a heatmap which shows, for each version, which URLs have been found in the code, and their corresponding location. The ones found in library code have cells colored in light blue, while URLs found in application code are represented by dark blue cells.

Figure 4.5: Stringoid Heatmap example.

Figure 4.5 shows the heatmap for the VLC Direct app: we can see that starting from version 92 the app includes new URLs related to the Google Mobile Services library. By checking the output of the libraries module we see in fact that release 92 adds the Google Mobile Services library to its code base.

4.1.7 API Evolution

Permissions are an abstract representation of the behavior of an app that users easily understand. However, many different operations are protected by the same permission, and the consideration of just permissions would miss relevant details. We thus extract the list of API calls that the app makes to the Android framework.

For this task we use Soot. The first step is to separate Android APIs from other method invocations and only keep the ones related to dangerous permissions. The process we used to create the list of APIs relative to each dangerous permission is explained in Section 4.1.8.

For each newly added API that requires a dangerous permission, whether it appears in the application code or in any third-party library, we save the API, the
list of available permissions, and the list of locations in which the API is used. We then parse the obtained APIs and extract the ones for which the required permission was already granted in a previous version.

4.1.8 Dangerous Permission API Mapping

Since in our analysis we focus on permissions that could potentially affect the user’s privacy, we want to consider only the APIs that require a dangerous permission. This task is not straightforward, as Google never released an official mapping between API calls and permissions, and the list might change with every Android major update. Given that a comprehensive mapping of API-permission does not exist, many researchers concentrated their efforts on generating one [33, 140, 141, 6].

To create our permission mapping we start from the one present in Androguard, which contains over 11,000 APIs. This information comes from the PS Scout’s automatically generated mappings [33]. It is used by the tool to identify which permissions are actually needed by the application’s code. We manually parse all the entries in the mapping, leaving only the ones that we are sure would actually require at least one permission, only leaving slightly more than 2000 APIs. We then parse all the mappings presented in DPerm [141], and add them to our set.

After that, for each permission, we manually search the web for how to implement basic functionalities using it (e.g. when analyzing the SEND_SMS permission we look for how to send an SMS). We add all mentioned APIs (such as the sendTextMessage method from the android.telephony.SmsManager class), and finally manually check for other APIs in the same classes in the official Android developers website (sendDataMessage, sendMultimediaMessage and sendMultipartTextMessage methods from the android.telephony.SmsManager class). Finally we inspect the source code of Android 6.0, and check all permission annotations and comments mentioning a permission. Our final mapping contains 2383 APIs.

4.1.9 Data Leak Analysis

Android apps usually access private user data like the list of contacts or current location. It is not clear whether the app leaks such sensitive information. To identify data leaks we resort to Flowdroid, the state of the art tool for static information flow analysis of Android apps [142].

The default set of sources (i.e. what sensitive data should be tracked) and sinks (i.e. program locations where a leak might happen) supplied with Flowdroid is very limited and does not include all APIs protected by dangerous permissions, significantly restricting the flows detectable by Flowdroid. In Cartographer we extend the given list of sources with APIs accessing sensitive information. Regarding sinks, we only consider APIs that send data to the Internet or store it...
to the file system, which can be a temporary storage enabling data leaks at later points in time.

Starting with the most comprehensive list of sensitive sources and sinks as provided by Droidsafe [143] (8,500 sources, 3,700 sinks), we gradually remove unrelated APIs, such as ones from the following categories:

- UNMODELED
- NFC
- SYNCHRONIZATION_DATA
- DATABASE_INFORMATION
- BLUETOOTH_INFORMATION
- GUI

Afterwards, we ensure that our final selection includes all relevant dangerous APIs introduced in Section 4.1.8.

Flowdroid produces data flows in the form:

\[ source_{method_0} \leadsto sink_{method_1} \]

where source and sink are Android API methods; method_0 and method_1 are API call site locations. We leverage the API–permission mapping defined in Section 4.1.8 and library analysis from Section 4.1.3 to characterize these flows. We translate them to

\[ PERMISSION_{loc_1} \leadsto SINK_{loc_2} \]

pairs, where loc can be either ‘in-app code’ or a particular library name.

Most of the data in the Android system can be accessed via a dedicated component, the Content Provider, which provides APIs to query, insert, and update data. The data accessed is specified by providing a special URI string when invoking the Content Resolver at runtime: for example, access to contacts in address book is done by passing the ContactsContract.Contacts.CONTENT_URI string as method argument. To know which data is being accessed, we need to get the value of the URI parameter.

In order to map ContentResolver data types, we need to create a list of all possible URIs used in the Android platform, which require a dangerous permission. We obtain it by manually parsing all the classes in the android.provider package from the official Android documentation 9. As a result, we substitute content resolver API sources with two types of URI values. The first one is represented by constant strings starting with the “content://” prefix. The other one is a Uri object which name contains the “CONTENT_URI” string. Thus, we use

---

these values as new data sources and only consider the data flows that contain content resolver APIs.

Flowdroid is unable to report the actual URLs and file paths of sinks; thus, we extend it to capture this information. As these values are usually passed—sometimes incompletely—into helper methods for further assembly, we resort to *inter-procedural analysis* to extract the precise sink destinations.

We tackle this problem with data flow analysis. APIs that create network connections or write to files are considered to be sinks. Conversely, all constant strings matching URL or file path patterns are treated as data sources. The resulting data flows $URL_{flow}$ link URLs to `openConnection` methods (and file paths to `write` invocations) in a context-sensitive manner. For each data flow path we search for $URL_{flow}$ with the longest common path suffix. For instance, the flows:

<table>
<thead>
<tr>
<th>Source Method</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Sink Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>source_method0</td>
<td>node1_method1</td>
<td>node2_method2</td>
<td>sink_method3</td>
</tr>
</tbody>
</table>

are transformed into:

<table>
<thead>
<tr>
<th>Source Method</th>
<th>Sink Method</th>
<th>URL Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>source_method0</td>
<td>sink_method3</td>
<td>sink+URL_method3</td>
</tr>
</tbody>
</table>

Static analysis is not able to identify all URLs and file names, as some of them are resolved only at runtime.

As Flowdroid performs static analysis, it might produce over-approximated results: some flows may be infeasible in practice. One example are flows whose sources require a permission which is not requested by the application.

In our analysis we distinguish two type of flows: the most interesting ones are the flows that access user sensitive data, which are protected by a dangerous permission, such as the list of contacts. The second type of flows contains all flows that access information which is either not protected by a dangerous permission, such as network settings, or flows that have a source for which we cannot derive the proper permission, such as unrecognized content resolver APIs. When we analyze flows of this latter type, we prefix them with the tag *NP* (*non-dangerous permission*).

### 4.1.10 Data Analysis

In the Data Analysis step we combine information from all the previous modules to provide a more in-depth analysis. For instance, we check whether the application layout changes when a new flow appears.

One example of combining results from different scripts is Figure 4.6, where we combine data gathered from the FlowDroid and App Info tasks: in the heatmap we show on the y-axis the data flows of the application, enhancing the data with information regarding the status of the permission needed by the flow source according to the following legend:
Chapter 4. Cartographer: Analyzing the Evolution of Android Apps

Figure 4.6: TripAdvisor Flow.

A  permission already asked in previous version;
N  permission newly asked;
- the flow does not require any dangerous permission;
M  permission missing;
R  permission revoked (it was requested in the previous version and then removed);
G  permission newly asked and automatically granted by Android because there is another permission in the same permission group already asked by the application;
S  special case for when we have multiple permissions to check at once (e.g. for a location flow we have to both check the ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION) and we do not fall in any of the previous cases, for example because one of the required permissions is revoked and at the same time another is added.

We can see that the READ_PHONE_STATE to INTERNET flow, which appears in version 160229020, has the READ_PHONE_STATE permission newly asked (N) in the first version. However, the permission is revoked (R) in the following version and never added back until a few releases later when the flow is no longer present. With this information we can understand that, despite finding the data flow with FlowDroid, it can only be exploited in the first version it appeared, as the required permission is no longer requested afterwards. This allowed us to discover that FlowDroid reports unfeasible flows, since the supposedly leaked data is protected by a permission which the application has not requested.
4.2 Empirical Study on Android Releases

We used Cartographer to run an empirical study on how Android apps change across different releases. We considered the following research questions:

- **RQ1**: *How does a new data flow correlate to other changes in the release?* To answer this question we analyze whether the UI layout changes when new flows are introduced. We also check the status of the permissions required by the flow source to understand if they are missing, appear for the first time together with the new flow, or were already requested in a previous version of the app. Finally, we want to understand if new flows are accessing new information or are leaking already accessed data throughout different sinks (for example, a new flow appears where data is sent over the network, but data from the same source was already being written to a file in the previous release).

- **RQ2**: *How do web domains relate to changes in the layout?* Similarly to what we do for information flows, we want to understand if the fact of connecting to new domains is related to some changes in the layout, or completely transparent to the user.

- **RQ3**: *How do information flows evolve during the lifetime of an application? How do third party libraries play a role into the app evolution?* We look for evolution patterns, such as if flows tend to be active during the whole analyzed period, or if they just last for a few releases only.

In the following sections we address these research questions and try to answer them with the data we extracted with Cartographer.

4.2.1 Dataset

For our empirical study we sampled 235 different applications available in Androzoo [71]. This repository provides unrestricted access to over 5.7 millions Android applications along with their metadata, allowing straightforward reproducibility of the research.

Our selection strategy followed two main objectives: 1) collect apps with at least 50 releases to have enough data for a study on the evolution of the app over time; 2) produce a representative distribution of apps. As previous literature showed that Google Play Store is a largely trustworthy [130, 131, 132] source of applications, we limited our dataset to the apps from this store.

The final dataset contains 14,880 releases, published within the timeframe of 21 August 2008 to 14 January 2017. The distribution of downloads per application is close to Normal, as shown in Figure 4.7. The majority of apps in our dataset (199) have a star rating of over 4 out of 5, with 30 applications having a star rating
of over 3, and just one having a rating of over 2. Finally, the dataset covers all
the 32 Android categories. It focuses on high quality and popular applications.
The number of releases per app spans from a lower bound of 50 up to 171, with
an average value of 63.

We conclude that the dataset is quite varied and representative.

**Dataset Statistics**

Before discussing the results of our study, we present some statistics on our
dataset. We found flows in 160 out of the 235 applications in our dataset (68%).
In total, Cartographer reported 68166 flow instances. We define as flow instance
the single path from a source to a sink reported by Flowdroid. A flow can com-
prise multiple flow instances. A single app can have multiple instances of the
same flow over several releases. The following is the list of the 10 most popular
flows (by number of flow instances):

- NP_INTENT $\rightarrow$ NP_FILE : 14628 (21.31%)  
- NP_INTENT $\rightarrow$ INTERNET: 11023 (16.06%)  
- NP_CONTENT_RESOLVER $\rightarrow$ NP_FILE : 9344 (13.61%)  
- READ_PHONE_STATE $\rightarrow$ NP_FILE : 5435 (7.92%)  
- ACCESS_LOCATION $\rightarrow$ NP_FILE : 4686 (6.83%)  
- NP_CONTENT_RESOLVER $\rightarrow$ INTERNET: 4583 (6.68%)  
- ACCESS_LOCATION $\rightarrow$ INTERNET: 4291 (6.25%)  
- NP_PACKAGE_MANAGER $\rightarrow$ NP_FILE : 3423 (4.99%)  
- READ_PHONE_STATE $\rightarrow$ INTERNET: 2730 (3.98%)  
- NP_PACKAGE_MANAGER $\rightarrow$ INTERNET: 1847 (2.69%)

Not all flows are proved to leak sensitive user data. For instance, the type of
the data determined by the NP_INTENT source depends on the intent payload.
This information is unavailable since Flowdroid does not support inter-component communication analysis. Similarly, NP_CONTENT_RESOLVER sources may query either application specific data or data which origin has not been recognized. All the flows in our study end up in either INTERNET or FILE sinks. We assume that files can be a temporary storage of leaked data, which may be sent out at a later point in time.

For private data disclosure analysis we only consider flows with sources that require dangerous permissions. In our study we have not detected significant number of flows originating in files with identified names. Therefore, we decided to exclude all flow with FILE sink. The following list contains all flows that leak sensitive user data:

- ACCESS_LOCATION $\rightarrow$ INTERNET: 4291 (54.67%)
- READ_PHONE_STATE $\rightarrow$ INTERNET: 2730 (34.78%)
- GET_ACCOUNTS $\rightarrow$ INTERNET: 805 (10.26%)
- READ_EXTERNAL_STORAGE $\rightarrow$ INTERNET: 23 (0.29%)

Unsurprisingly, the most common flows leak the user location and the device id number (whose access is respectively granted by the ACCESS_LOCATION and READ_PHONE_STATE permission), and send them over the Internet. Device id is often used as a unique identifier, and location is often used for geolocation services.

### 4.2.2 RQ1: How does a new Data Flow Correlate to Other Changes in the Release?

We identified 252 unique flows and discarded 28 of them, as the permission required to access the data was not requested by the application, making the data inaccessible. Out of the 224 flows left to analyze, 56 (25%) originated in libraries, and 168 (75%) originated in application code.

Since we are interested in new flows only, we discarded flows that were already present in the previous release of the app. Due to this filter, we were left with 202 new flows. Out of those, 141 (69.80%) were flows leaking data from a new source, while the remaining 61 (30.20%) leak data from an already leaked source.

We analyzed 202 flows that we found, and compared the permissions and UI layouts with their state in the version before the flow was introduced. We found out that only 30 (14.85%) sources are protected by newly asked permissions, whereas 170 (84.15%) new flows require a permission that has been already granted previously. The remaining 2 belong to the special case described in Section 4.1.10.

This means that either leaked data have already been used inside the application and started being leaked in a following release, or that developers add over-privileged permissions from the start for later use.

The comparison of UI layouts after a new flow has been added showed that the layout changed in 167 out of 202 cases (82.67%). Conversely, in about one
fifth of all cases the addition of a new flow was completely invisible to the user with no visible change in UI. This might be a serious privacy threat if the new flow is accessing information protected by a permission which has been already granted in a previous release.

4.2.3 RQ2: How do Web Domains Relate to Changes in the Layout?

Similarly to what we did for flows, we analyze the status of the layout when there is a connection to a new domain, either identified statically or dynamically.

For some apps we miss information regarding layout changes for some releases, because the layout folder created by the Apk Content Analysis is missing. This could happen for example because the application interface is entirely in HTML and it is loaded in WebView.

We found out that for the dynamic analysis out of 57,183 new domain connections, 62.7% of the times (35,844) there is a layout change, 29.8% (17,066) there are no changes, and 7.5% (4,273) the Apk Content analysis did not generate a layout folder.

We observed similar results for the 20,742 URLs identified by Stringoid, with the proportion more in favor of layout changes: 16,606 (80.1%) layout changed, 4,015 (19.3%) layout unchanged and 121 (0.6%) unknown.

We checked all the domains with VirusTotal and analyzed the ones reported as malicious by at least 3 VirusTotal sources, ending up with 16 domains. We manually analyzed these domains checking the category reported by VirusTotal’s Forcepoint ThreatSeeker. We performed a whois lookup to understand if it would be legit for the application to connect to that domain. We report the list of elements which could be malicious in Table 4.1.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>app.wapx.cn</td>
<td>malicious web sites, mobile malware</td>
</tr>
<tr>
<td>hotgirls.glkx.gdn</td>
<td>malicious web sites</td>
</tr>
<tr>
<td>byprizes.party</td>
<td>elevated exposure</td>
</tr>
<tr>
<td>byprizes.party</td>
<td>elevated exposure</td>
</tr>
<tr>
<td>900t.21045.xyz</td>
<td>elevated exposure</td>
</tr>
<tr>
<td>dn2.apphale.com</td>
<td>uncategorized</td>
</tr>
<tr>
<td>dnsseed.bitcoin.dashjr.org</td>
<td>suspicious content</td>
</tr>
<tr>
<td>shouji.360tpcdn.com</td>
<td>potentially unwanted software</td>
</tr>
</tbody>
</table>

We further analyzed the applications that connect to the reportedly malicious domains. We noticed that Hola Launcher and Dolphin Browser have a common subset of malicious domains they connect to. We initially hypothesize that the root cause could be a shared library, but our analysis found only three libraries in
common: Google Mobile Services, Android Support v4 and Facebook. As these three libraries are widely used, and as we did not discover malicious behavior in other apps using those libraries, we conclude that the connection to such malicious domains is not due to the use of suspicious libraries in the app.

4.2.4 RQ3: How do Information Flows Evolve During the Lifetime of an Application?

We collected the most common flow patterns (using a 1% frequency threshold) in Table 4.2. We use 1 to report the presence of flows and 0 for their absence.

The reported patterns count a total of 802 flows, out of which 676 (84%) had both a source and a sink inside library code, while the remaining 126 (16%) had at least one of them in the app code. Count and Frequency columns refer to flows in general, while Lib Freq. and Appcode Freq. report the frequency of that pattern in library and app code flows, respectively.

We can see that while third party libraries tend to increase the number of flows, the number of flows originating in app code remains fairly constant.

While analyzing the heatmaps generated by Cartographer, we came across some interesting flow evolution, which we report and comment in the remaining part of this section. The heatmaps we present have the same style and features as the one presented in Section 4.1.10, where we combine the information on flows together with the permissions required by the application.

![Figure 4.8: Yahoo Aviate Launcher Flow.](image)

Figure 4.8 shows the dangerous flows of the Yahoo Aviate Launcher app.
There is a strange pattern regarding the location: at some point, in version 18829, all flows disappeared.

Figure 4.9 shows all the dangerous flows of Noom Coach. There are many new flows in version 134, all associated to previously granted permissions. In version 186 there is a new flow LOCATION to INTERNET, and the changes in layout are very small.

The flows of Security Master app are shown in Figure 4.10: we can see three vertical stripes of flows that, on consecutive releases, appear and disappear. Such a strange pattern led us to further investigation: we discovered that those versions declare, in groups of two, the exact same `versionName` in the manifest. We suppose that in this case we have two applications that were build for various Android versions, with different libraries and code, but that were both fetched by Androzoo.

In Figure 4.11 we can see the flows of the PETATTO CALENDAR app. There is a flow in the Flurry library that shows how the location leaks to an Internet sink. Such flow, however, does not have the necessary permission to access the user’s location, since such permission was added in version 92, just before the flow disappears in the next release. The Flurry library is however not removed, which means that either the lib has been updated, or FlowDroid cannot identify the flow anymore.
We encountered 59 flows requiring dangerous permissions that originate in libraries: the most common sources for those flows are reported in Table 4.3. This shows that flows in libraries follow the same general trend reported in Section 4.2.1: the most commonly leaked sources are related to ACCESS_LOCATION and READ_PHONE_STATE permissions.

Table 4.3: Most common flow sources libraries.

<table>
<thead>
<tr>
<th>Flow</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_LOCATION</td>
<td>63%</td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>31%</td>
</tr>
<tr>
<td>READ_EXTERNAL_STORAGE</td>
<td>2%</td>
</tr>
<tr>
<td>GET_ACCOUNTS</td>
<td>2%</td>
</tr>
</tbody>
</table>

The libraries in which we found more flows are two advertisement and monetization libraries, MoPub (32%) and Inmobi (19%). Both libraries only contain flows leaking the user location, and the results are confirmed by [50], which reports the two libraries in the top 10 of the most installed in the dataset they analyzed.

4.2.5 RQ4: How do API Calls Evolve?

The API evolution analysis is based on the data extracted as explained in Section 4.1.7. For this analysis we rely on 228 applications analyzed, as for 7 packages we experienced a crash while extracting the APIs.

We found 1047 APIs for which the related permission is requested in the same version as the API is added. Excluding APIs found in obfuscated code, 797 (64%) APIs are found in library code, and 413 (36%) in application code.

For 9,360 newly added APIs, the relative permission was already requested in a previous version. After removing APIs we found in obfuscated classes, 8,259 (91%) APIs are added in libraries, and 800 (11%) in the application code.

From our data we can see that it is 9 times more frequent to have the permission already asked when a new API is added. We also found out that most of the
newly added APIs (88%) related to dangerous permissions are added in libraries.

4.3 Parental Control Apps Analysis

We use Cartographer to analyze 419 releases of 46 parental control applications, focusing our study on how developers access sensitive data. This analysis is part of a larger study, which also covers the analysis of Privacy Policies, Terms of Service, and a dynamic analysis of the applications. In this thesis we report only the part related with the static analysis performed by the author of the thesis.

4.3.1 Dataset

Google Play does not provide any specific category for parental control apps. Thus, to find potential parental control apps we search for the string “Parental control app” on Google Play’s search engine. It is important to note that we repeated this search at different points during the duration of the project to account for newly published software, following always the same approach. For the results of this search query, we eliminate manually any app that does not implement parental control functionalities. We ended up with 61 parental control apps. Interestingly, the majority of these parental control apps are classified as Tools (41.7% of apps) by their app developers, despite the presence of the more specific Parenting category (15% of apps).

Android app developers often release new versions to include new features to their software or to patch previous vulnerabilities [3]. Such changes may have an impact on the privacy impact of these apps [116]. To obtain 429 available versions for the list of 61 apps we crawl APKPure 10, a public dataset which contains historical versions of android applications. We discard those prior to 2016, as we deem them too old for our study. Our dataset finally contains 419 versions from 46 parental control apps.

For each app we harvest metadata publicly available on the Google Play store, including app descriptions, number of downloads, user ratings, privacy policies, and developer details. We use this metadata in several parts of our analysis to

---

10https://apkpure.com/
11https://apkpure.com/
contextualize the results of our empirical analysis, as detailed in each relevant section. Furthermore, while we do not claim that the 46 apps in this study is an exhaustive list of every available parental control app, in the rest of this Chapter we will show that this is a varied and complete list from a popularity and developer type standpoint.

**Apps’ popularity.** The number of installs per app—a metric used to infer a given app’s popularity by previous research papers [144, 145]—varies widely: at one extreme 10 apps have more than 1M downloads, while at the opposite extreme 17 apps have less than 100k downloads. When considered together, the 46 apps have been installed by more than 22.2M. The most downloaded apps tend to have a similar user rating (on average 3.5 out of 5) than those with a lower number of installs (parental control apps with less than 100k have a mean rating of 3.2 out of 5). Anecdotally, we found that several app developers justify this fact in their app descriptions claiming that low ratings come from children that are unhappy because the app works well.

**App developers.** We extract the apps’ signing certificates to identify the set of companies (or individuals) releasing parental control apps. We find that most developers are behind one single parental control app, except for 3 developers that have released two different apps. Also, 13 (21%) developers publish other unrelated software in Google Play besides the identified parental control apps. The most remarkable cases are *Yoguesh Dama*, an Indian company which has published 38 apps of different types (for volume control, screen locking, wallpaper, and more), and *Kid Control Dev* a Russian company which also builds apps unrelated to children such as flashlight apps.

One may assume that the use of parental control apps is mostly predicated on trust. Therefore, we investigate whether the identity of the app developer plays a role in the parents’ choice. Our departing hypothesis is that parents might prefer to trust software developed by well-known security company like *McAfee* or *Norton*. However, we find that these developers only account for 8.7% of total apps, and they are not the most installed ones—only one of them reaches 1M installs. The most popular parental control apps are those developed by companies specialized in kid-related software, specifically (e.g. *Kiddoware* and *Screen Time Labs*). Most parental control apps seem to monetize their software through in-app purchases and monthly licenses, yet they typically offer a free-trial period. According to Google Play metadata, 23 of them contain in-app purchases for unlocking features and buying licenses, which can range from less than 1 Euro to 350 Euros.

**Delisted apps** We note that 10 applications were removed from Google Play since we collected our list of apps two years ago. The reason why these apps have been delisted is unclear. The removal could be triggered by the developers (e.g. companies no longer operating), or by Google Play’s sanitation process (Play

---

12[https://kiddoware.com/](https://kiddoware.com/)
13[https://screentimelabs.com/](https://screentimelabs.com/)
Protect \(^{14}\)). We still analyze these apps, and report when they can not be tested because they do not longer works.

### 4.3.2 Permission Analysis

Android implements a permission model to control application’s access to personal data and sensitive system resources, as discussed in Section 1.2. Besides Android’s official permission list defined in the Android Open Source Project — grouped in different risk levels, the most sensitive ones labeled as “dangerous” ones— any app can define its own “custom permissions” to expose its resources and capabilities with other apps. In some cases, these features are exposed by pre-installed applications \(^{146}\).

The analysis of the permissions requested by an app (and its evolution across different releases) offers a high level idea of what protected system resources and data the app has access to. We now study the permissions that parental control apps usually request, and whether there are major differences between monitoring and restriction apps.

#### Prevalence and Usage of Permissions

Figure 4.12 plots system permission requests for the apps in our dataset. The plot reports one app per row, and each row shows the requested permissions, listed on each column.

The top part of the plot shows the apps in the restriction category, and the bottom part lists the ones we classify as monitoring. Column-wise, we differentiate permissions according to their privacy risk level \(^{15}\): i) dangerous permissions (first block from left, in blue)—i.e., those that could potentially affect the user’s privacy or the device’s normal operation, and therefore should be explicitly granted by the user; ii) normal permissions (second block, in green)—i.e., those that access data or resources outside the app’s sandbox, and therefore are granted automatically by the system—; iii) signature permissions (third block, in orange)—i.e., those that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. We also render system permissions in the same category, some of which were deprecated in recent releases of the Android framework, but still work on older ones; and iv) custom permissions (last block, in yellow)—i.e., those that apps define to protect their resources. Due to space constraints we only show custom permissions belonging to Google apps.

Most apps in our dataset have more than one release. Figure 4.12 uses gradients to show the percentage of releases of each app requesting a specific permis-

\(^{14}\)https://www.android.com/play-protect/

\(^{15}\)We refer to the official documentation for Android 6 and 7, the most popular platforms in 2019.
Chapter 4. Cartographer: Analyzing the Evolution of Android Apps

Figure 4.12: Permission heatmap. We list one app per row, indicating the permissions it requests. We differentiate dangerous permissions (blue), from normal (green), signature (orange), and custom (yellow).
Darkest colors show that all releases of the app request the permission.

Figure 4.12 suggests that parental control apps need lots of permissions in order to offer their service. Considering the median value, parental control app in our dataset request 27 permissions, 9 of them being labeled as dangerous. For comparison, the top 150 apps from the Google Play Store in May 2019 request 18 permissions, 5 of them labeled as dangerous. In fact, we find that the extreme parental control app in our dataset (com.nationaledtech.Boomerang) requests 91 permissions, of which 16 are dangerous; much more that typically considered privacy invasive non-parental control apps (such as com.facebook.orca, which requests 59 permissions, 16 of which are dangerous).

We see that the over ∼80% of parental control apps, regardless of their category, request access to location, contacts, and storage. Popular non-dangerous permission requests across parental control apps are related with Android’s package manager (monitor installed apps), and to control system settings. Despite the fact that our attention focuses primarily on dangerous permissions, it is important to highlight that non-dangerous permissions can be harmful too. One example is the BIND_ACCESSIBILITY_SERVICE permission, which is requested by 11 apps of the monitoring apps (115 releases), and 1 of the restriction apps (8 releases) in our dataset, and can be used to monitor and control the UI feedback loop and take over the device [9]. The permission is also reportedly used to perform an ad fraud, where developers generate fake clicks without the user noticing.\footnote{https://www.theregister.co.uk/2017/12/11/google_accessibility_crackdown_paused/}
Despite not being explicitly flagged as “dangerous”, also some of the deprecated system permissions are worth noticing, since they protect low level functionalities such as the ability to retrieve the list of running processes on the device. We observe that during runtime these apps have special instructions for the user to accept such permission, as their ability to monitor the children’s behavior depends on it.

At the other side of the spectrum, we notice that 2 parental control apps do not request any dangerous permission. One of them (com.redgreentree.app) replaces the Android default launcher to let the child use only apps approved by the parent.

We analyzed the other app (com.saferkid.parentapp) and discovered that the child’s app is not available in the Google Play store, which explains why it does not ask for any dangerous permission, but must be instead directly downloaded from the company’s website. This is the only app for which we detected this behavior.

Anomalous permission requests

A number of applications leverage dangerous permissions that are rarely used by other parental control apps. We analyze all apps that request permissions that less than 10% of apps request, considering monitoring and restricting apps separately. To identify whether these permissions are justified, we search for any information on their privacy policy that may justify their use. Two restriction apps result anomalous since they request permissions to send sms, process outgoing calls and read contacts. We found justification for one of them in the description. Regarding the monitoring category, we identify 7 apps as anomalous, as they request permissions to receive wap push, record audio and read calendar. 3 of them no longer appear on the Google Play (com.kakatu.launcher, com.appobit.familyorbit and com.bitdefender.parental2013). The remaining 4 do not justify why they request these anomalous permissions.

Table 4.4 lists those cases that we found insufficiently transparent in their app description and privacy policies.

Permission requests across app releases

Previous studies show that Android apps tend to request more permissions over time [1, 122], and we confirmed the trend in our study presented in Chapter 3. Interestingly, we observe the opposite behavior in the parental control ecosystem. App com.sentry.kid, developed by LidrTech Ltd., decreases its number of dangerous permissions from 19 to 11 in its released published on Jan 2nd 2018. We observe similar changes in com.calmean.parental.control (it decreases from 16 to 9 dangerous permissions, an later increases to 13 again), com.infoweise.parentalcontrol.secureteen.child (decreases from 11 to 6 permissions), and com.netspark.mobile (decreases from 5 to 0).

Restriction vs. Monitoring apps

Figure 4.12 reveals that monitoring apps tend to request more dangerous
Table 4.4: Unusual permission requests. We report whether the apps have been unlisted from Google Play (GPlay column).

<table>
<thead>
<tr>
<th>App</th>
<th>GPlay Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.pdlp.android.app</td>
<td>PROCESS_OUTGOING_CALLS SEND_SMS</td>
</tr>
<tr>
<td>com.whisperarts.kidshell</td>
<td>READ_CONTACTS</td>
</tr>
<tr>
<td>com.appobit.familyorbit</td>
<td>✓ READCALENDAR</td>
</tr>
<tr>
<td>com.mmguardian.childapp</td>
<td>RECORD_AUDIO</td>
</tr>
<tr>
<td>com.bitdefender.parental2013</td>
<td>✓ RECEIVE_WAP_PUSH</td>
</tr>
<tr>
<td>com.sentry.kid</td>
<td>READCALENDAR RECORD_AUDIO</td>
</tr>
<tr>
<td>com.symantec.familysafety</td>
<td>RECEIVE_WAP_PUSH</td>
</tr>
<tr>
<td>com.fsp.android.c</td>
<td>READCALENDAR</td>
</tr>
<tr>
<td>com.kakatu.launcher</td>
<td>✓ RECORD_AUDIO</td>
</tr>
</tbody>
</table>

permissions than restriction ones. Figure 4.13 compares the distribution of dangerous permissions requested by each type of parental control app. The difference between monitoring and restriction apps is notable even for the rightmost permissions in Figure 4.12, and in particular regarding browser controlling permissions. As expected, monitoring apps access the browser bookmark history more than restriction apps. However, restriction apps rely more heavily on custom-specific browser permissions to control the browsing activity on the child’s device, as we explain in the rest of this Section.

**Custom Permissions**

Android gives developers the opportunity to define custom permissions in order to expose parts of the functionalities of an app to others apps [32]. We find 28 custom permissions that judging by their name have been declared by parental control app developers (e.g. de.protectyourkid.permission.WRITE_SETTINGS). We did not render those permissions in Figure 4.12 for clarity reasons. Examples of this behavior contain permissions from companion apps like the Spin browser that substitutes the default user browser or the companion parent app (e.g. com.safekiddo.parent). These permissions are used from the children app to access or control functionalities of such companion apps.

Nevertheless, we also find custom permissions declared by different developers, like com.pdlp.android.app using custom permissions from de.protectyourkid. This hints to agreements across developers who take advantage of functionality from other apps when both are installed on the same device. We also find several
apps using custom permissions related to phone manufacturers [146]. The most frequent permissions are Samsung-related, used by various releases in 21 apps of our dataset, then follow Huawei (10 apps), HTC (8 apps), and finally Sony and Oppo (4 apps). These vendor-declared permissions allow app developer to gain access to other system features not available through the official Android Open Source Project. In some cases, as in the case of the Samsung KNOX API, app developers must become Samsung partners to gain access to their proprietary APIs. Furthermore, the user never gets to accept or reject custom permissions as they are automatically granted without showing a warning to the user. Although com.dblp.android.app declares permissions belonging to these 5 of the manufacturers, the general trend for our dataset is to declare permissions for either one vendor or none of them.

4.3.3 Third-party Library Analysis

While the majority of parental control apps request sensitive and dangerous permissions, the functionality protected by such permissions may be piggybacked by embedded third-party libraries whose purpose is providing advertising, analytics and other data-driven services.

We thus focus our attention on what libraries are embedded in parental control apps, and whether developers comply with their Terms of Service (crawling the ToS version corresponding to the release date of the app in April 2019, specially for those libraries that prohibit their usage in children-oriented software, such as Branch and Appboy (now rebranded as Braze). Furthermore, Google released in May 2019 a list of advertising libraries that are suitable for showing ads in children-oriented applications. The ad networks present in this list have self certified that they comply with the pertinent legislation (COPPA and GDPR).

We start our analysis by looking at all the releases of all the apps available in our dataset to see and classify what third-party libraries are mostly used by parental control apps. For that, we leverage LibRadar [53]. Then, we manually 1) analyze LibRadar’s output to sanitize the results, and 2) map packages to specific libraries and organizations. As the version of LibRadar we use can report multiple package names for the same library (e.g. com/paypal/.../onetouch and com/paypal/.../data), we cluster all packages belonging to the same library provider, ultimately identifying 157 unique SDK libraries. Finally, we classify libraries by their purpose by manually investigating each package reported by LibRadar, identifying to which library it belongs, and classifying it into its category. We could not identify the provider, and consequently the purpose, of 44

[18] https://branch.io/policies/#terms-and-conditions
[19] https://www.braze.com/privacy/
Chapter 4. Cartographer: Analyzing the Evolution of Android Apps

Table 4.5: Classification of third-party libraries found in parental control apps. For each category we report the number of libraries (the total number of package names matching one of these libraries) and a short description of the category.

<table>
<thead>
<tr>
<th>Category</th>
<th>Lib #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Social Network</td>
<td>1 (60 pkgs)</td>
<td>Social networks like Facebook</td>
</tr>
<tr>
<td>Advertisement</td>
<td>7 (82 pkgs)</td>
<td>Advertisement</td>
</tr>
<tr>
<td>Development</td>
<td>56 (582 pkgs)</td>
<td>Development support tools</td>
</tr>
<tr>
<td>Functionality</td>
<td>29 (172 pkgs)</td>
<td>App features (e.g. animations)</td>
</tr>
<tr>
<td>Support</td>
<td>43 (443 pkgs)</td>
<td>Support libraries (e.g. DB)</td>
</tr>
<tr>
<td>Analytics</td>
<td>21 (220 pkgs)</td>
<td>Analytics services</td>
</tr>
<tr>
<td>Unrecognized</td>
<td>44</td>
<td>Unidentified libraries</td>
</tr>
</tbody>
</table>

Most third-party libraries embedded in parental control apps are tools for development support—e.g. general purpose development libraries, JSON parsers, UI support libraries, etc—followed by libraries offering analytics services—e.g. Crashlytics, Flurry and AppsFlyer. The latter category of libraries are more concerning for parental control apps given their data-driven behavior and business models which may use children’s data for data-driven purposes. Therefore, for the rest of the study, we focus on those providing social network integration, advertisement and analytics services. Figure 4.14 shows the usage of libraries belonging to those categories: we can see that, while some of them are used by only a few releases, Google Ads, Google Firebase and Google Analytics are used by over 50% of the apps in our dataset, followed by Facebook SDK at 43%. Furthermore, some libraries belong to the same parent company, this is the case for Alphabet services (Crashlytics, Firebase, Google Analytics and Google Ads), which are present in 93% of the apps in the dataset when grouped together. This indicates that several companies have a bigger presence in the ecosystem, considering that it collects different data from several applications.

Apps Violating Libraries’ ToS

Following the findings by Reyes et al. that showed how several apps targeting children, used third-party libraries whose Terms of Service (ToS) prohibit their libraries mainly due to code obfuscation. The resulting categories are shown in Table 4.5, where we show for each category how many identified libraries it contains, reporting in parenthesis the total number of package names matching one of these libraries, and a short description of the category.

[21]https://abc.xyz
integration in children apps [147], we check whether parental control apps present a similar behavior. We manually check the terms of service of 5 libraries identified by Reyes that are also present in our dataset in April 2019. We find that a number of them changed their ToS since the study performed by Reyes et al. Both AppBoy and Branch still indicate that they should not be included in applications directly directed to children. Yet, leveraging a static analysis method based on Soot and FlowDroid [136, 142], we verify that two apps in our dataset (com.fsp.android.c and com.qustodio.qustodioapp) still integrate them in the latest version available in our dataset (November 2018). Furthermore, we study whether the libraries used for serving ads to children belong to the list of certified libraries released by Google. Out of the seven ad-related libraries that we find in our dataset, only two (AdMob and Unity) are currently certified by Google as suitable for minors.

**Who Handles Sensitive Data and What for**

Third party libraries can piggyback on the app permissions requested by the host app to gain access to sensitive data, and in some cases, they even force developers to request specific permissions. To have higher confidence on the permission usage and whether a permission is being requested by the app or by a third-party library, we statically analyze the bytecode of each app looking for Android API calls that actually use the permission. Next, we identify whether API calls are being used in the application code or in third-party library code. For this analysis we focus on the most recent release of each app in our dataset.

Table 4.6 summarizes the results of this analysis. For each dangerous permis-
Table 4.6: Attribution of permission usage to code belonging to application or third-party libraries. For each dangerous permission we report whether it is used only in the app code, library code or in both. The last column reports uses that we cannot attribute for sure to app or libraries.

<table>
<thead>
<tr>
<th>Permission</th>
<th>% app only</th>
<th>% lib only</th>
<th>% app &amp; lib</th>
<th>% unclear</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_COARSE_LOCATION</td>
<td>36</td>
<td>12</td>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td>ACCESS_FINE_LOCATION</td>
<td>33</td>
<td>9</td>
<td>55</td>
<td>3</td>
</tr>
<tr>
<td>CAMERA</td>
<td>14</td>
<td>7</td>
<td>7</td>
<td>72</td>
</tr>
<tr>
<td>GET_ACCOUNTS</td>
<td>25</td>
<td>12</td>
<td>4</td>
<td>59</td>
</tr>
<tr>
<td>PROCESS_OUTGOING_CALLS</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>88</td>
</tr>
<tr>
<td>READ CALENDAR</td>
<td>33</td>
<td>0</td>
<td>0</td>
<td>67</td>
</tr>
<tr>
<td>READ_CONTACTS</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>92</td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>39</td>
<td>26</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>READ_SMS</td>
<td>19</td>
<td>0</td>
<td>6</td>
<td>75</td>
</tr>
<tr>
<td>RECEIVE_MMS</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>75</td>
</tr>
<tr>
<td>RECORD_AUDIO</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>60</td>
</tr>
<tr>
<td>SEND_SMS</td>
<td>27</td>
<td>27</td>
<td>18</td>
<td>28</td>
</tr>
<tr>
<td>WRITE_CALL_LOG</td>
<td>12</td>
<td>12</td>
<td>0</td>
<td>76</td>
</tr>
<tr>
<td>WRITE_CONTACTS</td>
<td>22</td>
<td>11</td>
<td>0</td>
<td>67</td>
</tr>
<tr>
<td>WRITE_EXTERNAL_STORAGE</td>
<td>6</td>
<td>41</td>
<td>47</td>
<td>6</td>
</tr>
</tbody>
</table>
sion we report whether it is used only in the application code, only in the library code or in both of them. We have an additional column reporting the percentage of uses we cannot confidently attribute for reasons such as: 1) the permission is requested but not used, and therefore we could not find any corresponding API call; 2) our analysis misses relevant API calls because of incomplete information in the mapping between permissions and API calls; 3) the app has obfuscated code that we cannot attribute to a specific library/application.

We can see that a significant number of dangerous permission-protected methods seem to be invoked only by third-party libraries. In other cases, libraries may piggyback the set of permissions already requested by the host app. This suggests that users’ personal data might be disseminated and processed for usages beyond those offered by the app. Identifying the purposes of the data collection and processing of these third-party libraries is beyond the scope of this study. For these cases, we believe that app developers include permissions only because the library has this requirement. Unfortunately, for the permissions with high percentage of unclear uses we do not have clear results from this analysis because of the limitations of static analysis. We thus resort to dynamic analysis to overcome these limitations.

4.4 Limitations and Threats to Validity

Cartographer inherits all the limitations from the tools that it includes in its tool chain.

The limitations of FlowDroid come from the static analysis it implements: it does not consider paths involving asynchronous calls, and it does not trace inter-component flows. We could overcome those issues by integrating IC3 [148] into Cartographer. Moreover, FlowDroid is not sound, since it does not deal with reflection nor with native code.

Obfuscation plays an important role both in library analysis and APK content analysis. If the code is obfuscated it is not possible to directly compare code and resource names, so the layout analysis would report a big change between releases even when changes are actually minor. Moreover, if the code is obfuscated and LibRadar cannot identify a library, we cannot identify it with our analysis either.

All modules that implement static analyses only focus on Dalvik bytecode and ignore native code. This causes again unsoundness.

Cartographer suffers from limitations due to dynamic analysis too. It cannot fully explore applications, as Monkey cannot produce all the necessary inputs [113]. Cartographer can only observe behavior triggered by used events, and misses whatever might be triggered by environment factors, for example by timing.

When we aim to identify libraries using package name heuristics to overcome the limitations of LibRadar, we might incorrectly consider some application code
as library code.

Last but not least, we may in general have missed relevant information, since sometimes the tools we use might crash on specific APKs.

### 4.5 Takeaways

In the previous study, presented in Chapter 3, we report some problems with the permission model requesting permission at install time. In this chapter we present a more in-depth study which takes in consideration different features that change between consecutive releases of Android applications. Our goal is to have a better understanding of changes between consecutive Android releases, and better assess the validity of the permission model, highlighting eventual new issues we come across during our work.

The results of the study show that, for the majority of instances when developers add an API which requires a dangerous permission, the permission was already requested in a previous version of the app. This can make it hard for users to spot changes in the app behavior, especially since the changes that are easier to spot are the ones in the permission list, together with changes in the user interface.

Our study reports in fact that about 20% of newly introduced data leaks are added without any change in the permission list or in the application UI, being virtually invisible to the user and creating a serious privacy threat.

The findings presented in this Chapter confirm our hypothesis that the permission model previous to the release of Android 6.0 is flawed. We could in for example show that developers could exploit a permission granted in a previous release to start leaking data protected by that permission in a later version of the app.

Finally, we show that the current implementation of parental control applications can put childrens privacy at danger in diverse ways, and we hope that out study can raise awareness on the risks brought up by these applications.
Possible Threats of Automatically Granted Permissions

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

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

This chapter brings the following contributions:

- It reports for the first time a real threat for the privacy and security of Android users due to permissions groups.

- It presents an empirical study on 2,865,553 Android binary files, and shows that in a representative app store over ∼17% of the apps request danger-
Chapter 5. Possible Threats of Automatically Granted Permissions

ous permissions that the OS automatically grants without explicit user’s approval.

- We manually inspect some of the most suspicious cases, and we report clear abuses of apps that leak sensitive data such as user’s accurate location, list of contacts, history of phone calls, and emails.

The Chapter is structured as follows: Section 5.1 clearly present the threat scenario of automatically granted permissions. Section 5.2 presents our empirical study on the prevalence of automatically granted permissions in two large datasets. Section 5.3 shows the results of our preliminary static analysis to understand if apps actually use automatically granted permissions. Section 5.4 discusses in details the most interesting cases of clear information leaks. Section 5.5 reports the limitations and threats to validity for our study.

5.1 Threat Scenario

In the early releases of the Android OS, users had to accept all permissions requested by an app upfront to install and use it. Moreover, users had to approve every new app release before installing it on their device if the update included any new dangerous permission request. For years the research community heavily criticized this permission model, and pointed out its main flaws: app developers could easily request more permissions, and users were willing to grant them in order to use the app. Moreover, most users did not understand the implications of granting dangerous permissions to apps [7]. Thus, asking users to explicitly approve each release involving a change in the list of requested permissions had two effects: either users approved the change without worrying about the implications, or they did not approve it, keeping an old version of the app on the device, and augmenting the challenges for app developers who had to assume users could use different releases of their product.

Given a daily or weekly frequency of updates and potential changes in the list of permission requests, most app developers had the habit of asking for extra permissions in the first releases. The rationale was that even if early releases were not using some of the requested permissions, the user had to grant them anyway to use the app, and developers could use these permissions in later releases without bothering the user for further approval. Several studies showed that this practice caused many apps to be overprivileged, and left space for potential security attacks [6].

With the release of Android 6, Google significantly revised the permission model of Android apps, as reported in Section 1.2. Apps now dynamically ask for permissions, i.e. they ask for user consent when they use the permission for the first time. This change has the positive effect that users can now understand when and how an app needs to access a protected resource, and can decide to deny such
permission. Secondly, to simplify the permission model for user understanding, Android now uses permission groups. With this change, users would be asked to grant a permission only if the app asks for a permission that does not belong to any permission group that was already approved before. This means that, for instance, if an app already had the granted permission to access the coarse location of the user, it may access in later releases the fine-grained location information (obtained through GPS) without further consent.

While introducing permission groups clearly improves usability, making them preferred by the majority of users [27] over the previous model, it introduces more threats for their privacy and security, since behavior changes are more subtle to notice. Some empirical studies show that apps tend to leak more information over time, even when there is no change in the list of permissions [3, 116]. Other studies show that apps can leak sensitive information protected by dangerous permissions even without asking for that permission [40]. We argue that, on top of the risks already highlighted by other studies, permission groups pose a significant risk for the security and privacy of users, since developers can push a new release to the user’s device and may access resources that were not explicitly granted.

5.1.1 Threats of Permission Groups

The recent changes in the permission model affecting Android 8 and 9, reported in details in Section 1.2.1, are an indicator that the previous model was too permissive. Allowing apps with READ_PHONE_STATE to make phone calls, or to read or write the call log, as still allowed in Android 6 and 7, is extremely risky, and very counter-intuitive for users.

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

As an example, in order to provide the weather forecast for the current location any weather app would require to access the user’s location. Since the location does not have to be accurate, the app can initially ask for the ACCESS_COARSE_LOCATION, which does not use the data coming from the GPS. At any point in time, however, with a new update the app can suddenly retrieve the exact location of the user with the ACCESS_FINE_LOCATION using the GPS, and leak this information for unclear reasons. This change would go unnoticed for the user, since both permissions belong to the same group, and given that the user already agreed to share her location, the Android framework would not report that the app suddenly starts accessing the data from the GPS.
Chapter 5. Possible Threats of Automatically Granted Permissions

A more serious threat would come with any application with the permission to read SMS (RECEIVE_SMS), which apps often use to automatically activate an account registration via SMS. Given that the RECEIVE_SMS permission belongs to the SMS group, these apps in later releases could start reading, writing, and sending SMSs to premium numbers, effectively stealing money from the user.

5.1.2 Motivating Example

To properly explain the threat scenario, we use a real open source application as motivating example. The BirthDayDroid app from F-droid ¹, whose interface we show in Figure 5.1, helps users remember the birthdays of their contacts. To this end, the app scans the contact list on the device to collect birthday dates. It uses this information to show the age of each contact, and the days left until the

---

5.2 Prevalence of Automatically Granted Permissions

We aim to assess how Android applications use (or abuse) permission groups in the new permission model. Specifically, we aim to assess if and how often apps request permissions that the Android system would automatically grant, and we aim to analyze how they use such permissions. For this study, we need multiple releases of the same app, and we need to focus only on releases that use the new permission model, i.e. that target the Android platform 6.0 or above, as explained in Section 1.2.

For each app binary, we check the list of requested permissions, and we com-
Figure 5.3: Android release distribution of our dataset (ApkPure and AndroZoo)

We compare it with the closest previous release of the same app in our dataset. For each newly requested permission, we verify whether any permission in the same group is present in the previous release, and we mark it as automatically granted by the underlying system. This, essentially, simulates the scenario described with the BirthDayDroid app in Section 5.1.2, where users first install a release of the app, grant all the permissions it requires, and later receive an update of the app with a new list of permissions that the operating system grants without asking for any further user’s approval. We then further analyze all releases that include at least one automatically granted permission, and we assess if the app indeed accesses the information protected by the such permission, and if so whether it leaks this sensitive information.

We describe how we collect the dataset for our study in Section 5.2.1, and we explain how we design the study to quantify the risks of automatically granted permissions in Section 5.2.2. We finally present the results in Section 5.2.3.

5.2.1 Dataset Selection

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

Since crawling the Google Play store for several releases would take quite a
long time, we excluded crawling the official store. Although a recent study [54] reports that the Google Play API can be queried for lower version codes, we could not reproduce this technique.

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

ApkPure is a third-party store that constantly adds new releases and removes outdated ones (it keeps at most 30 releases per app, available to users). We consider all categories defined in the store, excluding only games. We exclude them since binaries of this category are usually very large, as they include a lot of graphics and data files. Their size can be up to 2GB, and they significantly impact the scalability of a large empirical study. For each other category we crawl apps with publication date later than the official Android 6 release (05 October 2015). This is how we retrieve 216,492 APKs related to 13,109 different apps for later analysis.

The largest fraction of our dataset comes from Androzoo [71], a repository of Android apps maintained by the University of Luxembourg containing over 9.6 million APKs. AndroZoo is the result of many years of efforts of crawling the Google Play and similar Android stores to download as many apps as possible for research studies. For our purpose we consider only apps crawled from the Google Play Store, with a release date subsequent to the Android 6 release, and with at least two releases available. Following this process we retrieve 3,274,192 APKs, related to 794,795 apps.

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

5.2.2 Experiment Protocol

Android applications are distributed via APK package files, which are essentially zip format-type archives. For our study, we need to extract the list of automatically granted permissions of each release, stored in AndroidManifest.xml file along with other metadata.

Despite being a simple and efficient analysis, downloading each APK, unpacking it and parsing the manifest of over 3 million apps would have been extremely time consuming. Our preliminary study showed that the Manifest file is usually located at the beginning of the archive. Thus, instead of downloading the whole package of each APK, which could be even hundreds of megabytes large, we initially download only the first 100Kb of the package. Next we parse the partially downloaded binary file using local file headers, extract AndroidManifest.xml if available, and decode the manifest file into a normal XML file with help of An-
droguard\textsuperscript{2}. We then parse the XML file and collect the target release of Android SDK, version name and version code, the list of app components and the list of requested permissions. We finally compare this list with the permissions requested by the closest previous release. If the APK requests any new permission that the Android OS would automatically grant, based on the target Android release, and the permission group presented in Section 1.2, we download the whole binary for further analyses. We discard it otherwise.

Downloading the first 100KB of the APK file does not guarantee that we download the Android manifest. Indeed, out of the $\sim3$ million APKs that we analyze with this heuristic, we could not analyze $\sim12\%$ of the APKs. We discard them for simplicity, obtaining a final dataset of 2,865,553 APKs.

Figure 5.3 plots how our aggregated dataset is split into different target Android releases. For each fraction we report the Android release that apps target (i.e. retrieved looking at the Android manifest), and the absolute number of APKs for the ApkPure and AndroZoo datasets (first and second number in parenthesis). For our analysis we discard APKs targeting releases prior to Android 6, since they use the old permission model.

5.2.3 Results

Our analysis reports that on Androzoo there are 55,442 applications with at least one automatically granted permission. Specifically, we flag 63,970 single APKs with at least one automatically granted permission (our dataset includes more than one APK per application). This amounts to $\sim7\%$ of apps of the whole dataset of Androzoo of the apps targeting Android 6 or above.

These numbers are even more worrisome for the ApkPure dataset. Our analysis reports that there are 2,135 applications, and 2,834 single APKs with at least one automatically granted permission. This amounts to $\sim17\%$ of apps of the whole dataset of ApkPure of the apps targeting Android 6 or above.

Figures 5.4 and 5.5 give an insights of the results of our analysis on the two datasets. Each plot shows the amount of automatically granted permissions on the right-hand side, and the enabler permissions (on the left-hand side) that appear in the previous release. Some permissions may be granted automatically due to the presence of more than one permission in the same group. We count both in the plot.

For instance, the plot shows that for the ApkPure dataset there are 148 applications that initially request the `ACCESS_COARSE_LOCATION` permission, and subsequently ask the `ACCESS_FINE_LOCATION` permission, which allows the app to access a more accurate localization, and obtain it automatically without further user’s approval. Some permissions that are automatically granted may be enabled by different permissions in the same group. This is the case, for in-

\textsuperscript{2}https://github.com/androguard/androguard

69
stance, for WRITE_CONTACTS, which is automatically granted in 72 apps in ApkPure, with 104 enablers. Roughly half of these apps initially request access to READ_CONTACTS, and later request and automatically obtain the permission to write the contacts list, while the remaining apps initially requested the GET_ACCOUNTS permission, from the same group.

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

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

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

5.3 Actual Use of Automatically Granted Permissions

Given the prevalence of automatically granted permissions, as shown in Section 5.2, we want to assess if apps actually use these permissions and how. We thus statically analyze the bytecode of the application to detect uses of automatically granted permission. We now explain how we implement this analysis, and we later present the results.

5.3.1 Static Bytecode Analysis

We resort to Soot [136] and FlowDroid [142] to statically analyze the whole package at the bytecode level to assess if the app actually uses each automatically granted permission. Android apps can access permission-protected resources in three different ways, all of which are covered by our analysis:

1. by calling specific Android API methods—e.g. access the user location with getLastKnownLocation();
2. by querying databases with ContentResolver—e.g. retrieve the contact list using the query("content://contacts") method;

3. by sending or receiving Intents—e.g. make a phone call by sending the android.intent.action.CALL intent.

Specifically, we parse the bytecode and look for invocations of methods of the Android API that are permission-protected. There are a few permission-methods mappings available from previous studies, such as PScout [33], Axplorer [140], DPSPEC [141]. Unfortunately, they all cover old Android versions (at most Android 7). We thus combine the lists of Axplorer and DPSPEC as a basis, and extend it with new entries, which we found by analyzing the official Android documentation \(^3\) and the AOSP source code \(^4\). For each dangerous permission

---

\(^3\)https://developer.android.com/docs
\(^4\)https://github.com/aosp-mirror
we thus have a list of Android API calls, intents, and content resolver URIs, protected with this permission. We then use Soot to parse each instruction in the Dalvik bytecode, looking for invocations related to the automatically granted permissions.

In order to understand whether these instructions can actually be executed (i.e. some of them may belong to functionalities of third-party components that apps do not use), we perform a reachability analysis of such statements. Starting from the method containing the statement of interest, we traverse the callgraph with a backward analysis and collect all methods in the call chain.

The main android components (activities, services, broadcast receivers, and data providers), which are declared in the Android manifest, are the only entry points of the app. If a call chain starts from any of the life-cycle or callback methods of these entry points, we assume that the permission protected statement

---

Figure 5.5: Automatically granted permissions in AndroZoo

---

5We augment the callgraph constructed by FlowDroid with additional edges.
Chapter 5. Possible Threats of Automatically Granted Permissions

is reachable and can be used by the app.

Broadcast receivers, which are responsible for the interception of system broadcast messages, can be registered both in the code and in the Android manifest. If a receiver declares in the manifest an intent filter capturing a protected intent, we assume that it is further used, and thus the app accesses protected information bundled with it.

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

- If the app contains at least a statement protected by the corresponding automatically granted permission, and our analysis reports this code as reachable, we consider it confirmed.

- If the app contains a statement protected by the corresponding automatically granted permission, but our analysis does not consider it as reachable, we mark it as unconfirmed, as we want to be conservative in identifying protected accesses. Nevertheless, those statements might be still reachable, for instance, via inter-component communication.

- If the analysis does not find any statement protected by the permission under analysis, we mark the permission as unused.

5.3.2 Results

The results of the static analysis are summarized in Figure 5.6, where we report for each automatically granted permission the amount of confirmed, unconfirmed and unused instances. Since we do not see a significant difference in the two datasets for this analysis, we present the aggregated information from the ApkPure and Androzoo datasets.

The results clearly show that in more than half of the cases (lighter color of the bars) permissions are actually used by the application. Some permissions, such as reading SMS (75%) and phone numbers (84%), or accessing the precise location (74%), have much higher values. Additionally, write calendar (82%) and voicemail (100%) permissions have a very high confirmed frequency, however their low number of occurrences—less than 0.2% of the total combined—makes them less interesting for our analysis.

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

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

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

To this end, we perform a dynamic analysis of a subset of these apps, manually running the app and profiling it to understand its behavior, and finally manually inspecting the bytecode if needed. We discuss how we perform the analysis in Section 5.4.1, and we present the most interesting findings in Section 5.4.2.

5.4.1 Dynamic Analyses and Manual Inspection

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

We install the instrumented app on the device and manually use it extensively—aiming to exercise as many activities/behavior as possible. We do not resort to automated tools to explore the behavior of the app because we want to achieve a high behavior coverage, which is still a challenge for the state of the art tools [113].

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

Many of the apps we consider in our analysis require valid credentials to work: we generate new accounts whenever possible. For the remaining cases—for example banking apps, which require valid accounts we cannot generate—we only log behavior and network traffic generated from the screen we have access to, such as home and settings.

Given the manual effort, we manually execute only 400 binary files, and we manually inspect only tens of them.

### 5.4.2 Confirmed Data Leaks

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

**cn.xender**  Xender is a popular file and music sharing app with over 100M installs. It leaks the device serial number to four URLs: ac.mobileanapp,

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

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

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

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

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

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

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

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

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

In the following paragraphs we describe more in details the most suspicious apps we analyzed thoroughly:

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

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

net.devmain.callblock “Call Block — number blacklist”, according to its description, blocks unwanted incoming or outgoing calls with the support of variety of filters. The app requests the READ_CONTACTS permission since its early releases, as this information is intuitively necessary for the app to function. Release 62, however, requests in the manifest the GET_ACCOUNTS permission, and the OS automatically grants it.

This release effectively collects e-mails of specific domains (filtered by gmail and hotmail keywords) from the accounts data on the device, ciphers them, and sends this information to their servers. Almost all strings in the app (including URLs) are encrypted with DES algorithm, which—along with obfuscation—hampers the manual inspection. The dynamic analysis of the instrumented app allowed us to get the real values.

The information sent in the JSON request could be an authentication request: it contains auth_key, auth_token fields, and the data message with username and password keys. The value for username is always set to ‘net.devmain.callblock’—the package name of the app, while password contains the ciphered e-mail in Base64 format. The data is sent to the http://zxc-atari.rhcloud.com/rest/resource/zxcreg service, which is no longer reachable.

The app does this operation as a background task, triggered from the main screen, and continues until it gets a response of successful data transmission. While this operation runs in the background, there is NO exposed functionality of the app that could justify this behavior (e.g. authentication). In the User Consent data section of Terms and Conditions, published on their website, the developers state they “use cookies and device data to personalize content and ads”. E-mails may simply be used as identifiers for further in-app purchases, though the manners used for collections and transmission make the activity quite suspicious.
**com.adstick.goodhabit1** “Good Habit” is a quiz game for children, intended to teach good manners. Version 1.3, published on August 09th 2016, requests the GET_ACCOUNTS permission. Version 1.5, published only two days after (on August 11th), requests the READ_CONTACTS permission. Right after starting, the app retrieves all the entries in the contact list on the phone, and sends them to http://www.kotadiya.com:3004/. At the time of the analysis this domain is no longer reachable.

The manual inspection of the bytecode confirms that in the onCreate() lifecycle method of the main activity, the app establishes a socket connection with the server, checks if the contacts have been already uploaded, and then creates a separate background thread which queries all contacts data using the ContentResolver, storing it in a field. Afterwards, it bundles the contact data together with the IMEI number, and sends the package via socket connection to the remote server. Meanwhile, the foreground Activity of the app suggests to log-in with a Facebook account. The app’s description does not justify this behavior in any way, and the binary code clearly shows that the app steals user private data without any consent. This application has been removed from the Google Play store.

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

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

A threat to the validity of our work is that our datasets may not be representative of the entire ecosystem: the majority of apps we analyzed were taken from the Google Play Store, which has been shown to be largely trustworthy [130, 131, 132]. We limit this risk by having in our dataset all the apps listed in ApkPure, an alternative app store for Android.

A limitation we have is that both our static and dynamic analysis might produce false negatives by failing to detect uses of automatically granted permissions. This is why we consider the number of leaks found as a lower bound. Static analysis may suffer false negatives because the construction of the call graph is not sound, we do not analyze the native code, and the permission mapping we used may be incomplete.

Dynamic analysis have false negatives by definition. By manually exercising the app we cannot be sure that we explore the whole behavior of the application. Moreover, apps have non deterministic behavior that we did not observe. Finally, we used a proxy to intercept network traffic even when encrypted. Our approach would not recognize data leaks in case they were encrypted or tampered by the application before being sent to the server. Moreover, Android apps often use SSL pinning to communicate sensitive information only to trusted servers (thus not our proxy). In all these cases we miss the information.

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

5.6 Takeaways

Our previous studies, presented in Chapters 3 and 4, showed that the Android model implemented previous to the release of Android 6.0 contained several flaws. Even though many improvements have been introduced in the new permission model, the results presented in this Chapter show that the new permission model still contains possible threats to user’s security and privacy. The addition of permission groups does in fact allows developers to exploit a previously requested permission to add another one in the same group and having it automatically granted by the system without the user noticing it.

In this Chapter we present an empirical study on 2,865,553 Android apps evaluating the threat posed by permission groups to the privacy and security of final users. Our study shows that often apps request new dangerous permissions, and the Android OS automatically grants it. Our analyses show that apps actually use over 56% of the automatically granted permissions. Moreover, our manual inspection reveals clear abuses of apps that exploit auto-
matically granted permissions to access sensitive user data—such as the accurate location or the list of contacts—and leak them either to servers belonging to the company developing the app or to third parties.

Our results are partially confirmed by the most recent changes to the permission model, such as the introduction of the CALL\_LOG permission group introduced in Android 9, as described in Section 1.2.1. The change does effectively hinder the behavior of the keyboard app which uploads the call log information to a server, presented in Section 5.4.2, forcing it to at least show the permission request to the user.

Our study highlights a concrete new threat. We are aware that balancing usability and privacy in mobile devices is a challenging task, and we acknowledge the effort that Android engineers are putting in enhancing the permission model, as highlighted in Section 1.2.1. However, as our study shows, these are not enough.
Conclusions & Future Work

This last chapter presents the conclusions of the thesis: Section 6.1 discusses our ideas for future studies, and Section 6.2 summarizes our findings and considerations on the analysis performed.

6.1 Future Research Directions

Android is constantly changing and evolving, and this may impact our analysis. Some of the changes mentioned in Section 1.2.1 would require us to update Cartographer to take them into consideration, as they might have an impact on the results of the analysis we perform. For instance, performing the analysis described in Chapter 5 using a different permission model could produce a different set of applications with automatically granted permissions.

We plan to enhance Cartographer by implementing the possibility of performing the analysis using different permission models, having the tool automatically picking the correct one for each APK based on the target Android API declared in the manifest.

While we mostly focused our analysis on the set of dangerous permissions, there exist other permissions, generally belonging to the signature group, that could allow an app to perform operations that could harmfully impact other apps, the operating system, or the user. While performing the empirical study on parental control applications, we came across many of those permissions, and we would like to perform an in-depth analysis on those permissions to see how and by which apps they are used.

Finally, it would be beneficial to extend our tool with the analysis of the behavior of an application with respect to its description, relying on existing work done in the area [120, 149].
6.1.1 Automated Android Test Update

Many researchers focused on the automatically produced inputs to explore Android apps, as reported in Section 2.1.7. Being able to exercise and test an application — achieving a high code coverage— in a completely or partially automated way is very important for both industry and research purposes. It would in fact both allow i) companies to test more extensively their applications, lowering their development costs and creating a higher quality product, and ii) researcher to scale their dynamic analysis to bigger datasets, especially in situations such as the study we presented in Chapter 5, where dynamic analysis is necessary to confirm the presence of data leaks.

The problem with the current state of the art tools is that, since there are millions of Android apps, it is tough to create a general approach which can be applied to the majority of them obtaining good results in terms of activities explored or code coverage. This is also due to specific issues, such as apps requiring to go through a registration or login process.

Besides the state of the art tools to automatically generate inputs to explore the app under analysis, there are also tools that allow users to manually define test cases, and then provide a framework to execute them over and over. One example of those tools is UI Automator \(^1\), developed by Google. This way, it takes a significant amount of time and effort to set up a test, but once this is done it can be repeated multiple times with low effort. Those tools are quite popular in industry, where they can cut the development costs of a new app as they effectively reduce the cost of testing applications. The problem of those tools is that the tests need to be manually updated every time the application’s user interface changes.

It is however quite uncommon for apps to completely redesign their user interface, and the changes that are introduced by the majority releases are limited to small changes, such as changing the disposition of elements in an activity, or updating the behavior of a button.

Our insight is to use such tools to define the tests once, and then to develop our own technique to detect changes in the app’s UI and try to automatically update the tests to achieve the same coverage.

For example, if one release updates the ID of a button, and this causes one of the tests to fail as it does not find the element anymore, our tool would try to identify whether the button is still in the UI, and determine its new ID. At this point we can simply update the test by replacing the old ID with the new one, thus being able to use the same test even on the new release.

We considered a number of frameworks (UI Automator, Espresso, Appium, Calabash, Robotium, Kakao, Selendroid), and finally decided to use Appium for exploratory project. Besides being one of the most used Android testing frameworks, Appium also gives us the advantage of directly working via command

\(^1\)https://developer.android.com/training/testing/ui-automator
line, as opposed to other tools which require Android Studio to operate.

We performed a preliminary study, where we created tests for 5 applications with more than 50 releases, and collected the faults that caused any of the tests to fail. We then identified the most common types of exception and tried to target the easiest to fix. We started from the ID not found, which is triggered when the test is looking for an element which is missing on the current activity, for example because it has been removed or its ID has changed.

By just focusing on this one exception, we were able to overcome a number of test failures by updating the ID of the missing element in the tests and executing them again without raising an Exception anymore.

Our next steps would be to expand our tool to automatically fix more fault and then make it available to all researchers.

6.2 Conclusions

In this thesis we focus our research on the evolution of Android applications, analyzing it under different aspects and reporting multiple findings which we hope can be leveraged to allow the creation of better and safer mobile applications in the future.

The preliminary study on the evolution of permission requests in Android apps paved the way for the rest of the thesis. The results we obtained report some issues in the way developers deal with the permission model, as evidenced by duplicated and overprivileged permissions. We also show that changes in permission requests and app functionalities are not necessarily correlated.

Our second empirical study reports that applications tend to add information flows over time and, most importantly, that those changes are very difficult for users to notice: the flows can for example be added during an update which presents no changes in layout nor in the permission list. This requires to address the problem on different levels, for example requesting developers to mention any new functionality that adds accesses to sensitive data when documenting the changes in a new release. We also introduce Cartographer, a ready-to-use framework to analyze the evolution of Android applications, which we make available to the research community. Cartographer extracts and visualizes various information from APKs, and combines them to create a report on the evolution of analyzed apps.

In our third contribution, where we update Cartographer to focus our analysis on parental control applications, we report that those applications are on average more permission-hungry than the top 150 apps in the Google Play Store, and that the current implementation of many of them does indeed put the children’s privacy at risk in multiple ways.

\footnote{https://github.com/gorla/appmining}
In our last empirical study we analyze the threat automatically granted permissions pose to sensitive user data. Starting from the two datasets we took in consideration, Androzoo and Tacyt, which have a combined total of over 11 millions releases, we identify 3,558 releases, belonging to 3,180 apps, which have automatically granted permissions.

Our study highlights a threat in the handling of permissions in the Android system, which was already suggested in the literature, but never specifically analyzed in details. Our results confirm that the threat is indeed real and that some developers are actually exploiting the current permission model to gain access to sensitive user data, proving right the reasons that pushed us to perform the analysis in the first place.

We are aware of the high difficulty in balancing usability and privacy in mobile devices, and also of the effort that Android developers are putting in enhancing the system, as highlighted in Section 1.2.1, however we would suggest to dive deeper in this particular issue to provide users a higher grade of protection for their sensitive data.
This work was supported by the EU FP7-PEOPLE-COFUND project AMAROUT II (n. 291803), by the Spanish projects DEDETIS (n. TIN2015-70713-R) and SCUM (n. RTI2018-102043- B-I00), and by the Madrid Regional projects N-Greens Software (n. S2013/ICE-2731) and BLOQUES (n. S2018/TCS-4339).


