What did Really Change with the new Release of the App?

Paolo Calciati[♡♢] · Konstantin Kuznetsov[♣] · Xue Bai[♠♡] · Alessandra Gorla[♡]

[♡]IMDEA Software Institute, Spain
[♣]Saarland University, CISPA, Germany
[♢]Universidad Politécnica de Madrid, Spain
[♠]Beijing Institute of Technology, China

ABSTRACT

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 an empirical study 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.

KEYWORDS

Android, app evolution, behavior change

ACM Reference Format:
Paolo Calciati[♡♢] · Konstantin Kuznetsov[♣] · Xue Bai[♠♡] · Alessandra Gorla[♡]. 2018. What did Really Change with the new Release of the App?. In *Proceedings of MSR '18: 15th International Conference on Mining Software Repositories, Gothenburg, Sweden, May 28–29, 2018 (MSR '18),* 11 pages.
https://doi.org/10.1145/3196398.3196449

1 INTRODUCTION

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 now has numerous, fast evolving, competitive markets. 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 competitiveness of the market, ensures that popular Android apps are released on a monthly or even weekly basis.

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 changed 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 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.

This paper presents Cartographer, a ready-to-use framework for users and market managers to analyze the evolution of an Android application. Cartographer extracts and visualizes various information: 1) it shows how an app uses sensitive data, thanks to a customized static data flow analysis; 2) it aims to identify the list of third-party libraries that the app uses, even if obfuscated; 3) it extracts the network traffic to have a list of hosts the application talks to; 4) it statically extracts sensitive Android APIs the application uses. Cartographer runs these analyses separately and combines the results to create a comprehensive report on how the app evolved.

This is not the first paper that aims to analyze how Android apps evolve in their behavior. While many of the existing literature regarding Android application evolution focus on dangerous permissions [10, 26, 28], in this paper we take a wider approach to have a more in-depth understanding of what changes between different releases. We use Cartographer to empirically analyze 14,880 releases. Our dataset comprises 235 applications with at least 50 releases.

[1] http://gs.statcounter.com/press/android-overtakes-windows-for-first-time

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196449

Our analysis reveals that Android applications tend to have more leaks of sensitive data over time, in line with what Ren et al. reports [24]. Though, the growth is largely determined by third-party libraries. Our study also shows 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, and that the vast majority of data flows only exists for a limited number of versions.

The remainder of the paper is structured as follows:

- Section 2 presents the information that defines the domain of Android applications and their versioning.
- Section 3 presents our tool and explains its workflow.
- Section 4 reports our finding, characterizing the evolution of data flows we encountered in our dataset.
- Section 5 presents the related work.
- Section 6 contains limitations and threats to validity.
- Finally, Section 7 summarizes our findings.

2 THE ANDROID ECOSYSTEM

Android developers can distribute their apps through markets such as the Google Play Store. It allows users to easily access new app releases, as they can be downloaded by default on any mobile device.

Different Android releases of the same app can be distinguished thanks to two identifiers. The versionCode is an automatically generated integer used for internal versioning. This number can 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 app release version that is visible to the user. The second identifier is versionName string. Most developers define versionNames in the format `major.minor`, but such convention is not enforced. Moreover, versionNames are not necessarily unique across releases. This value has no purpose other than to be displayed to users.

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 [2]: those APKs share the same versionName.

Each application comes with a manifest file, which contains essential information about the app: the Java package name that uniquely identifies the app; 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.

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.

As of Android 6.0, its security model is based on a runtime permission mechanism. The two most important protection levels are *normal* and *dangerous* permissions. The first ones protect functionalities that pose very little risk to the user's privacy. If an app requests a normal permission, the system automatically grants it. Dangerous permissions protect data or resources that involve user's private information. The user has to explicitly grant these permissions to the app when requested. With the new permission model that Android introduced with API level 23, however, apps requesting a dangerous permission would have it automatically granted if the user already approved another permission from the same group.

3 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.

Calciati et al. [10] 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 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 [5] on top of Luigi[3]. 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

[2] https://developer.android.com/google/play/publishing/multiple-apks.html

[3] https://github.com/spotify/luigi

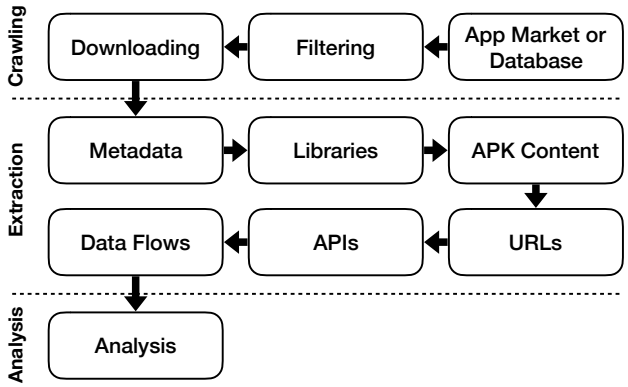

Figure 1: Main components and analysis flow of Cartographer

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 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.

3.1 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 implemented adapters for Androzoo [2] and for the F-Droid app store [4] 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.

3.2 Application Metadata

As described in Section 2, 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.

We use the Android Asset Packaging Tool (aapt) [5] 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 [6] 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 [7] 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.

3.3 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 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 [6, 18, 19]. In order to detect third-party libraries used in Android apps, we resort to LibRadar [19]. 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 [27] 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.

[4] https://f-droid.org
[5] developer.android.com/guide/topics/manifest/uses-feature-element.html
[6] https://github.com/androguard/androguard
[7] https://github.com/tdoly/apk_parse

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 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.

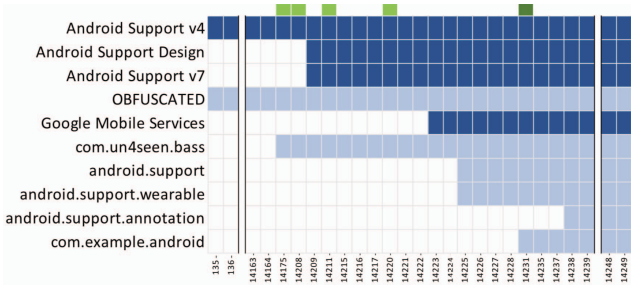

Figure 2: Libraries included in Smart Audiobook Player

3.4 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 [8] to extract resource files, contained in the *res* folder, including strings, drawable resources, and application layout files. In this step we also extract the application code files, contained in the *smali* folder.

We use this information to show which changes happened between releases at the code and layout levels, 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 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 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).

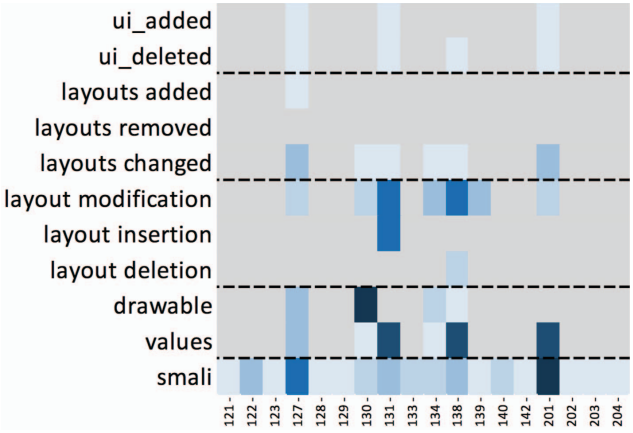

Figure 3: Apk Structure of Mobile Print - PrinterShare

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.

3.5 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 [11]. 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 [9] to extract the domains to which each APK connects and then analyze each of them on VirusTotal [10] 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

[8] https://ibotpeaches.github.io/Apktool/

[9] https://www.bro.org/

[10] https://www.virustotal.com

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.

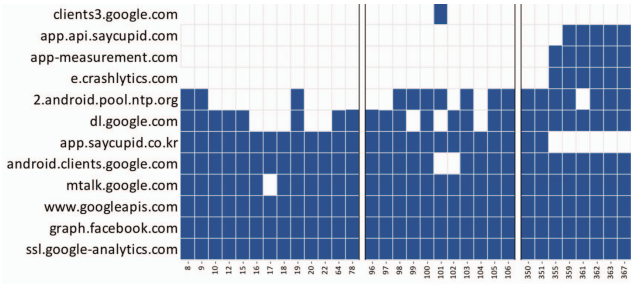

Figure 4: Saycupid Domains Heatmap

Figure 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).

3.6 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 Stringoid [23], 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.

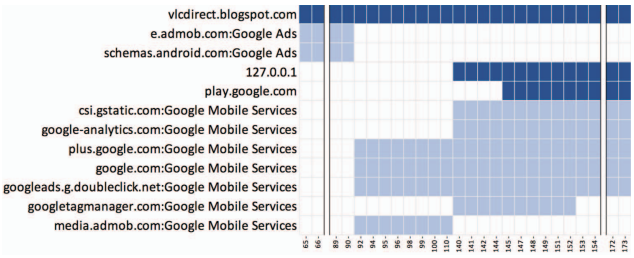

Figure 5: Stringoid Heatmap example

Figure 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.

3.7 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 3.8.

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.

3.8 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 [4, 7, 8, 13].

To create our permission mapping we started from the one present in Androguard, which contains over 11,000 APIs. This information comes from the PScout's automatically generated mappings [4]. It is used by the tool to identify which permissions are actually needed by the application's code. We manually parsed all the entries in the mapping, leaving only the ones that we were sure would actually require at least one permission, only leaving slightly more than 2000 APIs. We proceeded to parse all the mappings presented in DPerm [8], and added them to our set.

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

3.9 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 [3].

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 [14] (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 3.8.

Flowdroid produces data flows in the form:

$$source_{method_0} \rightsquigarrow sink_{method_1}$$

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 3.8 and library analysis from Section 3.3 to characterize these flows. We translate them to

$$PERMISSION_{loc_1} \rightsquigarrow SINK_{loc_2}$$

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 [11]. 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:

$$source_{method_0} \rightsquigarrow node1_{method_1} \rightsquigarrow node2_{method_2} \rightsquigarrow sink_{method_3}$$
$$URL_{flow}: \quad url_{method_4} \rightsquigarrow node3_{method_2} \rightsquigarrow sink_{method_3}$$

are transformed into:

$$source_{method_0} \rightsquigarrow sink_{method_3} \rightsquigarrow sink{+}url_{method_3}$$

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)*.

3.10 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.

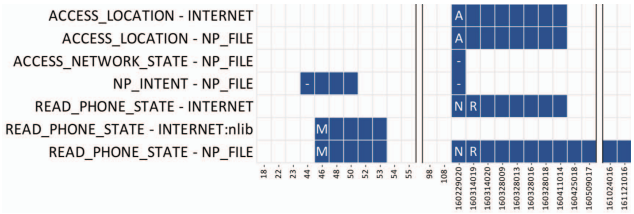

Figure 6: TripAdvisor Flow

One example of combining results from different scripts is Figure 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:

- 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;

[11] https://developer.android.com/reference/android/provider/package-summary.html

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 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 layout changes?* 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.

4.1 Dataset

For our empirical study we sampled 235 different applications available in Androzoo [2]. 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

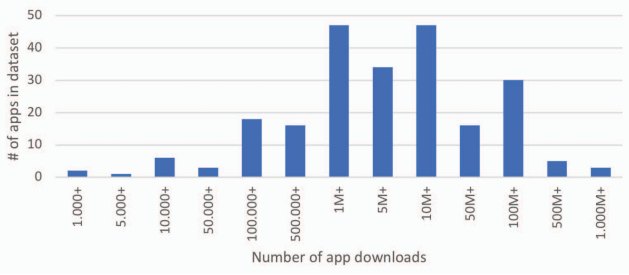

Figure 7: App Download Distribution

distribution of apps. As previous literature showed that Google Play Store is a largely trustworthy [1, 21, 30] 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 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 comprise 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 ⇝ NP_FILE : 14628 (21.31%)
NP_INTENT ⇝ INTERNET: 11023 (16.06%)
NP_CONTENT_RESOLVER ⇝ NP_FILE : 9344 (13.61%)
READ_PHONE_STATE ⇝ NP_FILE : 5435 (7.92%)
ACCESS_LOCATION ⇝ NP_FILE : 4686 (6.83%)
NP_CONTENT_RESOLVER ⇝ INTERNET: 4583 (6.68%)
ACCESS_LOCATION ⇝ INTERNET: 4291 (6.25%)
NP_PACKAGE_MANAGER ⇝ NP_FILE : 3423 (4.99%)
READ_PHONE_STATE ⇝ INTERNET: 2730 (3.98%)
NP_PACKAGE_MANAGER ⇝ 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 ⇝ INTERNET: 4291 (54.67%)
READ_PHONE_STATE ⇝ INTERNET: 2730 (34.78%)
GET_ACCOUNTS ⇝ INTERNET: 805 (10.26%)
READ_EXTERNAL_STORAGE ⇝ INTERNET: 23 (0.29%)

Unsurprisingly, the most common flows leak the user location and the device id number (whose access is granted by the ACCESS_LOCATION and READ_PHONE_STATE permission, respectively), 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 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 (75.00%) 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 3.10.

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.

4.3 RQ2: How do Web Domains Relate to Layout Changes?

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 57183 new domain connections, 62.7% of the times (35844) there is a layout change, 29.8% (17066) there are no changes, and 7,5% (4273) the Apk Content analysis did not generate a layout folder.

We observed similar results for the 20742 URLs identified by Stringoid, with the proportion more in favor of layout changes: 16606 (80.1%) layout changed, 4015 (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 1.

Table 1: Potentially malicious domains

Domain	Category
app.wapx.cn	malicious web sites, mobile malware
hotgirls.gikx.gdn	malicious web sites
byprizes.party	elevated exposure
byprizes.party	elevated exposure
90ot.21045.xyz	elevated exposure
dn2.apphale.com	uncategorized
dnsseed.bitcoin.dashjr.org	suspicious content
shouji.360tpcdn.com	potentially unwanted software

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.

4.4 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 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 3.10, where we combine the information on flows together with the permissions required by the application.

Table 2: Most common flow patterns

Pattern	Count	Frequency (%)	Lib Freq. (%)	Appcode Freq. (%)
0+ → 1+ → 0+	314	39.15	36.24	54.76
0+ → 1+	168	20.95	21.75	16.67
0+ → (1 → 0)+ → 0+	107	13.35	13.16	14.29
0+ → (1 → 0)+ → 1+	80	9.98	11.25	3.17
(1 → 0)+	60	7.49	6.95	10.32
1+	12	1.50	1.78	0.00

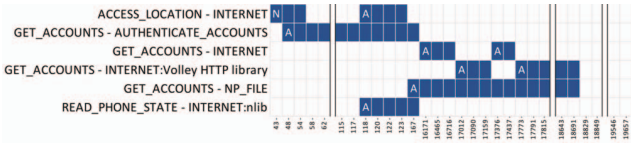

Figure 8: Yahoo Aviate Launcher Flow

Figure 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.

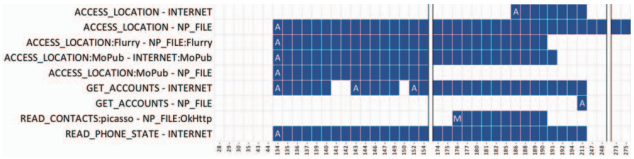

Figure 9: Noom Coach Flow

Figure 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.

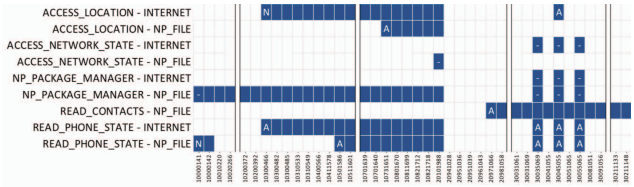

Figure 10: Security Master Flow

The flows of Security Master app are shown in Figure 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 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.

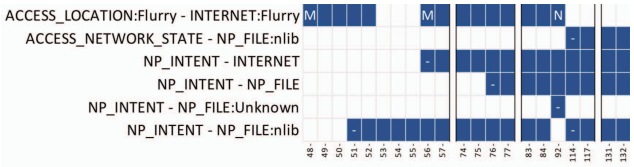

Figure 11: Data flows of PETATTO CALENDAR

We encountered 59 flows requiring dangerous permissions that originate in libraries: the most common sources for those flows are reported in Table 3. This shows that flows in libraries follow the same general trend reported in Section 4.1: the most commonly leaked sources are related to ACCESS_LOCATION and READ_PHONE_STATE permissions.

Table 3: Most common library flow sources

Flow	Frequency
ACCESS_LOCATION	63%
READ_PHONE_STATE	31%
READ_EXTERNAL_STORAGE	2%
GET_ACCOUNTS	2%

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 [9], which reports the two libraries in the top 10 of the most installed in the dataset they analyzed.

4.5 RQ4: How do API Calls Evolve?

The API evolution analysis is based on the data extracted as explained in Section 3.7. 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 9360 newly added APIs, the relative permission was already requested in a previous version. After removing APIs we found in obfuscated classes, 8259 (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.

5 RELATED WORK

There are already multiple publications regarding the evolution of Android applications. Wei et al. [28] 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. Krutz et al. [17] created a dataset of 4416 android releases, enhanced with information extracted using different static analysis tools. Taylor and Martinovic [26] presented a very broad study performed 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. With respect to the aforementioned papers, our study proposes a wider approach to the evolution of Android apps, as we do not focus only on the changes in permissions.

Ren et al. [24] analyzed the network traffic generated by 7665 releases belonging to 512 apps, and identified 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. With respect to our work, Ren et al. have a more in-depth dynamic analysis focusing on network traffic, while they do not consider any static analysis of the app code. Moreover, they analyzed roughly half of the releases we considered.

Calciati and Gorla [10] performed a study on over 14,000 applications, focusing on the evolution of permission requests. From their study it emerged that applications tend to add permissions over time, that many newly requested permissions are initially not used by the application code, and that when an applications removes a permission request it does not necessarily imply the removal of the corresponding functionality.

Hecht at al [16] presented PAPRIKA, a tool based on Soot and its Dexpler module to monitors 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.

Stevens et al. [25] analyzed permission usage and correlated it with StackOverflow questions regarding them, reporting that the likelihood of misusing a permission decreases with the popularity of the permission.

Zhang et al. [29] 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. [9] 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.

Derr et al. [12] conducted a study on libraries' updatability on 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.

6 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 [22] 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 [11]. 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.

7 CONCLUSIONS

In this paper we presented Cartographer, a ready-to-use framework to analyze the evolution of Android applications. Cartographer extracts and visualizes various information from APKs, and combines them to create a report on the evolution of analyzed apps.

The empirical study we conducted over 14,880 APKs using Cartographer allows us to report interesting evolutionary trends: we discovered that apps tend to add more data flows over time, even if the rate of growth is quite low and most data flows only last for a few releases. Another key finding is that for most of the added data flows which have sources protected by dangerous permissions, the application had already the permission requested in a previous version. We also found out that when developers add APIs that require a dangerous permission, most of the times the permission was already requested in a previous version. This can make it hard for users to notice changes in the app behavior, especially because the changes that are easier for users to spot are the ones in permissions and layout.

FlowDroid is able to identify data leaks within one component, and does not support inter-component communication analysis. In the future we plan to include the IC3 tool [22] in our framework to supply FlowDroid with the information about inter-component links. This would allow us to identify the information passed through intents, increasing the precision of Cartographer.

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 [15, 20].

The code of Cartographer is open source and available at:

https://github.com/gorla/appmining

ACKNOWLEDGMENTS

This work was supported by the EU FP7-PEOPLE-COFUND project AMAROUT II (n. 291803), by the Spanish project DEDETIS, and by the Madrid Regional project N-Greens Software (n. S2013/ICE-2731).

REFERENCES

[1] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. L. Traon. Empirical assessment of machine learning-based malware detectors for android - measuring the gap between in-the-lab and in-the-wild validation scenarios. *Journal of Empirical Software Engineering*, 21(1):183–211, 2016.

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. AndroZoo: Collecting millions of android apps for the research community. In *MSR 2016: 13th Working Conference on Mining Software Repositories*, pages 468–471, Austin, TX, USA, May 2016. ACM.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI 2014: Proceedings of the ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation*, pages 259–269, Edinburgh, UK, June 2014.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: analyzing the Android permission specification. In *CCS 2012: Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 217–228, Raleigh, NC, USA, October 2012.

[5] V. Avdiienko, K. Kuznetsov, P. Calciati, J. C. C. Román, A. Gorla, and A. Zeller. CALAPPA: a toolchain for mining android applications. In *WAMA 2016: Proceedings of the 1st International Workshop on App Market Analytics*, pages 22–25, Seattle, WA, USA, November 2016. ACM.

[6] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in android and its security applications. In *CCS 2016: Proceedings of the 23rd ACM Conference on Computer and Communications Security*, pages 356–367, Vienna, Austria, October 2016. ACM.

[7] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *USENIX Security: 25th USENIX Security Symposium*, pages 1101–1118, Austin, TX, USA, August 2016. USENIX Association.

[8] D. Bogdanas. Dperm: Assisting the migration of android apps to runtime permissions. *CoRR*, abs/1706.05042, 2017.

[9] T. Book, A. Pridgen, and D. S. Wallach. Longitudinal analysis of android ad library permissions. *CoRR*, abs/1303.0857, 2013.

[10] P. Calciati and A. Gorla. How do apps evolve in their permission requests? a preliminary study. In *MSR 2017: 14th International Conference on Mining Software Repositories*, pages 37–41, Buenos Aires, Argentina, May 2017. IEEE Computer Society.

[11] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, pages 429–440, Lincoln, NE, USA, November 2015. IEEE Computer Society.

[12] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes. Keep me updated: An empirical study of third-party library updatability on android. In *CCS 2017: Proceedings of the 24th ACM Conference on Computer and Communications Security*, pages 2187–2200, Dallas, TX, USA, October 2017.

[13] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS 2011: Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 627–638, Chicago, IL, USA, October 2011.

[14] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of Android applications in DroidSafe. In *NDSS 2015: 21st Annual Symposium on Network and Distributed System Security*, San Diego, CA, USA, February 2015.

[15] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *ICSE 2014: Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035, Hyderabad, India, June 2014.

[16] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien. Tracking the software quality of android applications along their evolution (t). In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, pages 236–247, Washington, DC, USA, November 2015. IEEE Computer Society.

[17] D. E. Krutz, M. Mirakhorli, S. A. Malachowsky, A. Ruiz, J. Peterson, A. Filipski, and J. Smith. A dataset of open-source android applications. In *MSR 2015: 12th Working Conference on Mining Software Repositories*, pages 522–525, Florence, Italy, May 2015. IEEE Press.

[18] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. Libd: Scalable and precise third-party library detection in android markets. In *ICSE 2017: Proceedings of the 39th International Conference on Software Engineering*, pages 335–346, Buenos Aires, Argentina, May 2017. IEEE Press.

[19] Z. Ma, H. Wang, Y. Guo, and X. Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *ICSE 2016: Proceedings of the 38th International Conference on Software Engineering*, pages 653–656, Austin, TX, USA, May 2016. ACM.

[20] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora. ARENA: An approach for the automated generation of release notes. *IEEESE*, 43(2):106–127, February 2017.

[21] Y. Y. Ng, H. Zhou, Z. Ji, H. Luo, and Y. Dong. Which android app store can be trusted in china? In *COMPSAC 2014: Proceedings of thehe 38th Annual International Computers, Software & Applications Conference*, pages 509–518, Västerås, Sweden, July 2014. IEEE Computer Society.

[22] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *ICSE 2015: Proceedings of the 37th International Conference on Software Engineering*, pages 77–88, Florence, Italy, May 2015.

[23] M. Rapoport, P. Suter, E. Wittern, O. Lhoták, and J. Dolby. Who you gonna call?: analyzing web requests in android applications. In *MSR 2017: 14th International Conference on Mining Software Repositories*, pages 80–90, Buenos Aires, Argentina, May 2017.

[24] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez. Bug fixes, improvements, ... and privacy leaks. In *NDSS 2018: 24th Annual Symposium on Network and Distributed System Security*, San Diego, CA, USA, February 2018.

[25] R. Stevens, J. Ganz, V. Filkov, P. Devanbu, and H. Chen. Asking for (and about) permissions used by android apps. In *MSR 2013: 10th Working Conference on Mining Software Repositories*, pages 31–40, San Francisco, CA, USA, May 2013. IEEE Press.

[26] V. F. Taylor and I. Martinovic. To update or not to update: Insights from a two-year study of android app evolution. In *ASIACCS 2017: Proceedings of the ACM Asia Conference on Computer and Communications Security*, pages 45–57, Abu Dhabi, UAE, April 2017. ACM.

[27] Vallée-Rai, Raja and Co, Phong and Gagnon, Etienne and Hendren, Laurie and Lam, Patrick and Sundaresan, Vijay. Soot – a Java Bytecode Optimization Framework. In *CASCON 1999: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 13–23, Mississauga, Ontario, Canada, Nov 1999.

[28] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the android ecosystem. In *ACSAC 2012: Proceedings of the 28th Annual Computer Security Applications Conference*, pages 31–40, Orlando, FL, USA, December 2012. ACM.

[29] J. Zhang, S. Sagar, and E. Shihab. The evolution of mobile apps: An exploratory study. In *DeMobile 2013: 1st international Workshop on Software Development Lifecycle for Mobile*, pages 1–8, Saint Petersburg, Russia, August 2013. ACM.

[30] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *NDSS 2012: 18th Annual Symposium on Network and Distributed System Security*, San Diego, CA, USA, February 2012.