ON UNDERSTANDING PERMISSION USAGE CONTEXTUALITY OF ANDROID APPS

MD ZAKIR HOSSEN

A thesis
in
The Department
of
Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science
in
Information Systems Security
Concordia University
Montréal, Québec, Canada

December 2018

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **MD ZAKIR HOSSEN**

Entitled: **On Understanding Permission Usage Contextuality of Android Apps**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science
in
Information Systems Security

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. Walter Lucia ______________________ Chair
Dr. Amr Youssef ______________________ Examiner
Dr. Emad Shihab ______________________ External Examiner
Dr. Mohammad Mannan ______________________ Supervisor

Approved ______________________
Chair of Department or Graduate Program Director

__________ 2018 ______________________
Dr. Amir Asif, Dean
Gina Cody School of Engineering and Computer Science

Abstract

On Understanding Permission Usage Contextuality of Android Apps

MD ZAKIR HOSSEN

In the runtime permission model, the context in which a permission is requested/used the first time may change later without the user's knowledge. Prior research identifies user dissatisfaction on varying contexts of permission use in the install-time permission model. However, the contextual use of permissions by the apps that are developed/adapted for the runtime permission model has not been studied. Our goal is to understand how permissions are requested and used in different contexts in the runtime permission model, and compare them to identify potential abuse. We present ContextDroid, a static analysis tool to identify the contexts of permission request and use. Using this tool, we analyze 38,838 apps (from a set of 62,340 apps) from the Google Play Store. We devise a mechanism following the best practices and permission policy enforcement by Google to flag apps for using permissions in potentially unexpected contexts. We flag 30.20% of the 38,838 apps for using permissions in multiple and dissimilar contexts. Comparison with VirusTotal shows that non-contextual use of permissions can be linked to unwanted/malicious behaviour: 34.72% of the 11,728 flagged apps are also detected by VirusTotal (i.e., 64.70% of the 6,295 VirusTotal detected apps in our dataset). We find that most apps don't show any rationale if the user previously denied a permission. Furthermore, 13% (from the 22,567 apps with identified request contexts) apps show behaviour similar to the install-time permission model by requesting all dangerous permissions when the app is first launched. We hope this thesis will bring attention to non-contextual permission usage in the runtime model, and may spur research into finer-grained permission control.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr. Mohammad Mannan for his constant support and guidance throughout my journey at Concordia University. He has been very supportive of this work and provided constant feedback that resulted in a conference publication.

I am grateful to all my labmates for being so nice and friendly to me. I learned a lot from everyone, especially, Xavier de Carné de Carnavalet and Dr. Lianying Zhao. I feel honoured to have worked with them.

I would like to dedicate this thesis to my parents who believed in me when things were not going well. Their support and prayers from the other side of the planet helped me concentrate on my research and courses.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation & Overview

The runtime permission model enables finer-grained control of resources. The new model was introduced in Android 6.0 to facilitate user decision by providing situational context (e.g., current state of the app representing the purpose of resource access) when the permissions are requested for the first time. An app can trick a user to grant a permission in a valid context, and then use it in malicious/unexpected contexts without the user's consent/knowledge. For example, accessing GPS when the user attempts to find the current location in a map is a valid context, but accessing GPS when the app is in the background may be unwanted. Indeed, such contextual differences may defy user expectations [21, 30, 31].

In contrast to the contextual analysis of resource access in the old install-time permission model [11, 16, 32], such studies in the runtime model are limited. Wijesekera *et al.* [30] modify an older version of Android to analyze contextual integrity of Android apps and conclude that users mostly rely on the surrounding context in which a permission is requested to grant/deny a permission [31]. In this work, we focus on regular apps that are developed/adapted for the runtime permission model and perform the first large-scale study to understand the contextual use of resources in the runtime permission model.

We develop ContextDroid, a static analysis tool that extracts the *context* when a permission is requested and used in an app using an app-wide call graph. We define a context based on the active User Interface (UI) component that is requesting the permission (and using it, if granted). To differentiate between user activities, we identify five Android components (`Activity`, `Fragment`, `Service`, `AsyncTask` and `Broadcast Receiver`) that represent different types of UI and functionality. We devise a mechanism for identifying potential abuse of permissions in unexpected contexts based on the developer policy and permission request best practices suggested by Google [2, 3].

There are several challenges in statically extracting contextual information from Android apps. For example, Android apps can be obfuscated using various obfuscation tools (e.g., ProGuard [12]) during the build process. While Android framework classes and methods are excluded from obfuscation, classes derived from support libraries that are shipped with the APK can be obfuscated by ProGuard (unless otherwise configured by the developer). We must identify *Fragments* and permission related APIs that are derived from the support libraries. We propose a combination of an extended call graph and sub-signature matching to identify the contexts that are otherwise obfuscated by ProGuard or similar tools.

Our evaluation reveals that apps often use permissions in multiple and dissimilar contexts. Overall, we flag 30.20% of the 38,838 apps for potential abuse: requesting permission in one context while using it in another context and using permissions in multiple contexts that include third-party libraries. 34.72% of the flagged apps (64.70% of the VirusTotal detected apps in our dataset) are also detected by at least one VirusTotal engine.

We find that very few apps show permission rationale, even if the user denied the permission when previously requested. Only 5% apps in our dataset show some kind of rationale if a permission was previously denied. 13% apps still show the typical behaviour of the install-time permission model by requesting all permissions at once during first launch in the new runtime permission model.

1.2 Thesis Statement

The primary goal of this dissertation is to evaluate changes in app behaviour in terms of permission usage in the runtime permission model. We want to compare permission usage contexts of apps with their request contexts and identify potential inconsistencies.

As part of this objective, we consider the following research questions:

Question 1 How often do apps use the same permission in multiple contexts?

Question 2 How often do apps use permission in dissimilar contexts?

Question 3 Can non-contextual use of permissions be attributed to hidden intent? In particular, is it possible to connect multiple and dissimilar usage contexts to malicious behaviour?

1.3 Contributions

1. We present ContextDroid, a static analysis tool that extracts the contexts in which the permissions are requested and the contexts in which they are used. ContextDroid extracts the call paths that lead to sensitive API calls associated with permissions and extracts contextual information. Our methodology for context identification may be useful for other studies.

2. We analyze 62,340 regular Android apps to understand contextual resource usage under the runtime permission model. To the best of our knowledge, this is the first study on contextual resource usage in the runtime permission model, involving apps that target only the new model.

3. We devise a methodology to identify potentially unexpected behaviour of apps following the recommended practices and policies for permission usage suggested by Google. Based on this mechanism we flag 11,728 (30.20% of the 38,838 apps) apps. Comparison with VirusTotal shows that non-contextual use of permissions can indeed overlap with malicious behaviour. Overall, 4,073 (34.72% of the flagged 11,728 apps) of the flagged apps are also detected by VirusTotal (i.e., 64.70% of the 6,295 VirusTotal detected apps).

4. Our analysis tool can be used by app market maintainers early in the vetting process to identify apps that may be violating user expectation and subject them to further analysis.

1.4 Related Publication

Conference Paper. The work discussed in this thesis has been peer-reviewed and published in the following conference:

> On Understanding Permission Usage Contextuality in Android Apps. Md Zakir Hossen and Mohammad Mannan. *Data and Applications Security and Privacy (DBSec'18)*, July 16 - 18, 2018, Bergamo, Italy. Lecture Notes in Computer Science, vol 10980. Springer, Cham.

In addition, I analysed companion Android apps of 11 smart toys for children. Our paper has been peer-reviewed and published in the following workshop:

> Towards a Comprehensive Analytical Framework for Smart Toy Privacy Practices. Moustafa Mahmoud, Md Zakir Hossen, Hesham Barakat, Mohammad Mannan, and Amr Youssef. *International Workshop on Socio-Technical Aspects in Security and Trust (STAST'17)*, December 5, 2017, Orlando, Florida, USA.

1.5 Outline

This thesis is organised as follows. Chapter 2 discusses the necessary background of Android and its permission models. Chapter 3 describes the related work on contextual resource usage in Android. Chapter 4 describes our methodology of defining and identifying request and usage contexts. Chapter 5 discusses our dataset, the performance of ContextDroid and the results of our analysis. Finally, Chapter 6 concludes our work.

Chapter 2

Background

In this chapter, we describe the necessary background of Android and its permission model.

2.1 Android Platform Architecture

Android is a Linux based open source platform designed for a wide range of devices including smartphones, smartwatches and smart TVs. The software stack of Android comprises of the System Apps, Java API Framework, Native libraries, Android Runtime, Hardware Abstraction Layer and the Linux Kernel. We briefly discuss the major components in this section. Figure 1 shows the platform architecture of Android.

2.1.1 System Apps

Android smartphones are pre-built with a set of apps that provide the most basic features (e.g., SMS). These apps can be replaced with third-party apps to provide similar or more functionalities. Moreover, they can be used by the third-party apps to perform specific tasks for them (e.g., using the default Camera app to take a photo).

2.1.2 Java API Framework

The application framework written in Java acts as the gateway to the underlying set of features provided by the Android OS. Apps can get access to the hardware sensors and native libraries through a large set of Java APIs.

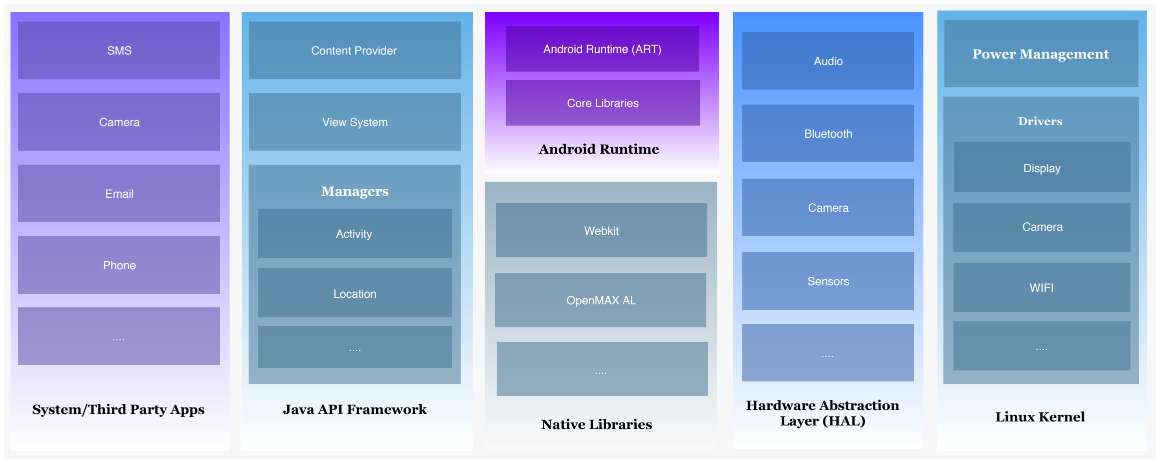


Figure 1: Android platform software-stack (from left to right)

It contains all the building blocks that are essential to develop a third-party Android app. This layer comprises of different modules for creating app views, managing data, resources and app life-cycles.

2.1.3 Native C/C++ Libraries

Many of Android's core system components are written in native C/C++. Android provides Java APIs that allow third-party apps to use some of the native functionalities. Moreover, if apps need to implement a part of their code in native C/C++, they can use the Android NDK. The NDK is particularly helpful for developers who want to reuse existing libraries of their own or from open source repositories. It can also be used for performance optimisation for computationally expensive operations. Android apps can also use some of the sensitive resources in the native code. For example, apps can record audio in native C++ through the JNI.

2.1.4 Android Runtime

This module includes the Android Runtime (ART) and a set of core libraries of Java programming language. ART was introduced in Android version 5.0. Older versions of Android had the Dalvik runtime.

Optimised ahead-of-time compilation introduced in ART improved the overall performance with stricter install-time verification in comparison to the older runtime (i.e., Dalvik). In addition, it provided more efficient debugging support and better garbage collection. In the ART, apps run in their own processes (each with a new ART instance).

2.1.5 Hardware Abstraction Layer

Hardware Abstraction Layer (HAL) provides interfaces that allow higher level Java APIs to access different hardware sensors. Vendors implement the interfaces based on their requirement without the need to modify anything in the top level architecture.

HAL decouples the top level application framework from the low level driver implementations. There are multiple library modules in HAL that belong to different hardware sensors. For example, one module in the HAL is responsible for implementing interfaces for the Camera sensor. When apps need to access a hardware sensor, Android loads the corresponding module.

2.1.6 Linux Kernel

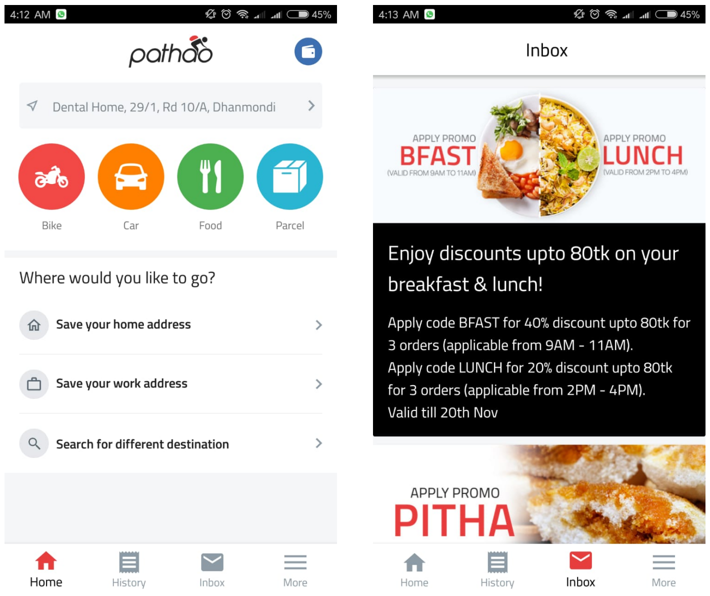
Android is built on top of the Linux kernel. It enables Android to inherit key security features of this well-known kernel. Android's Linux kernel helps manage critical low-level functionalities for the ART. This layer typically handles threading, power and memory management.

Linux kernel also contains the hardware driver implementations. Hardware vendors develop driver implementations in this layer based on the interfaces provided by HAL.

2.2 Android App Components

`Activity`, `Service`, `Content Provider`, `Broadcast Receiver`, `Fragment`, and `AsyncTask` are some of the key components of Android apps. These components act as the building blocks for Android apps. Apart from `Content Provider` that helps manage and share data between different apps, other components represent various elements associated with UI and events.

All these components have their own *life-cycle* methods that are invoked by Android OS and act as entry points for the components. These entry points can also be



Enjoy discounts upto 80tk on your breakfast & lunch!

Apply code BFAST for 40% discount upto 80tk for 3 orders (applicable from 9AM - 11AM).
Apply code LUNCH for 20% discount upto 80tk for 3 orders (applicable from 2PM - 4PM).
Valid till 20th Nov

(a) Home Fragment (b) Inbox Fragment

Figure 2: Example of an Activity that contains multiple fragments

the entry point of an app functionality depending on how they are implemented by developers.

2.2.1 Activity

`Activity` is the most prominent Android component. Third-party Android apps are built around one or more `Activities`. `Activity` provides a full-screen UI that enables users to interact with the app. Figure 2 shows example of an `Activity`.

Different screens of an app is implemented by different `Activities`. It has several *life-cycle* methods that are invoked in different situations. For example, *onResume()* is called when the `Activity` becomes visible in the screen and *onStop()* is called when the app goes in the background and is no longer visible on screen.

2.2.2 Fragment

`Fragment` defines parts of the GUI. `Fragments` can be considered as UI modules that represent part or full screen of an `Activity`. An `Activity` can hold multiple `Fragments`, and a `Fragment` can be reused in multiple `Activities`. `Fragment` has its own *life-cycle* methods similar to an `Activity`.

Multiple `Fragment`s inside the same activity represents different UIs and functionalities. Figure 2 shows two different `Fragment`s with different functionalities. Figure 2a allows users to search for rides or foods. Figure 2b represents an inbox for messages coming through the app. In this case, both the `Fragment`s reside inside the same activity although their purposes are different.

2.2.3 Broadcast Receiver

Broadcast Receiver receives updates from the OS whenever there is a change of state and can perform tasks without interacting with the UI. App developers can also implement their own *Broadcast Receivers* and broadcast an update to trigger a background task.

2.2.4 Service

A `Service` runs in the background without a UI. A `Service` can be initiated from any of the Android components including a `Service`. Although apps can check whether a permission is granted or not in a `Service`, they can't request for permissions. Apps can only request permissions from a foreground component. However, they can use the granted permissions in a `Service`.

2.2.5 AsyncTask

`AsyncTask` performs tasks in the background and communicates the results to the UI thread. `AsyncTask` can be initiated from any other component. It performs shorter operations in a background thread without freezing the UI and publishes the result back to the UI thread once done.

A task is performed in an `AsyncTask` in four steps. The first step is invoked in the UI thread to setup related tasks. In the second step, `AsyncTask` goes to a background thread to perform the task. In the third step, it publishes the progress of the task to the UI thread. Finally the forth step is invoked in the UI thread where the results of the task are passed.

2.3 Android Permission Model

2.3.1 Permissions

Android defines several levels of protection for device resources. Third-party apps can use *normal, signature* and *dangerous* level permissions. Each of these levels contain a set of permissions that protect specific resources. Android apps need to specify the permissions that they need in the AndroidManifest file that contains the meta data of the apps. When a resource protected by one of these permissions is accessed by an app through the Java APIs, Android first checks whether it has the corresponding permission before allowing access.

Normal permissions protect resources that are less privacy and security sensitive (e.g., BLUETOOTH). *Signature* permissions are mainly used by system apps. Third party apps can also define their own permissions. *Signature* permissions allow apps from the same developer to share their resources. The system decides to grant these permissions without user intervention if the app is signed with the same key of the app that defined the permissions. *Signature* level permissions include SYSTEM_ALERT_WINDOW, WRITE_SETTINGS etc.

Android maintains a set of *Dangerous* permissions that protect privacy sensitive resources. While individual dangerous permissions regulate access to specific actions or tasks (e.g., READ_PHONE _STATE), they are categorized/clustered into permission groups that protect specific resources (e.g., READ_SMS, WRITE_SMS are grouped into SMS).

2.3.2 Install-time Permission Model

Android used the install-time permission model up to version 5.0. In this model, all the specified permissions are granted if the app is installed in the device. Therefore, users had to agree to grant all the permissions at install-time without even running the app.

2.3.3 Runtime Permission Model

In the runtime model, dangerous permissions are requested at runtime ideally when the app really needs them. Android developer policy asks the developers to request permissions in-context [3]. Asking for a permission when an app feature really needs it makes more sense to the users, who may otherwise deny the permission.

2.3.3.1 Permission Request

Instead of showing prompt for each permission, Android requests permission for the permission group. For example, if an app requests RECEIVE_SMS permission, the system does not show that the app needs RECEIVE_SMS permission. Instead, it shows request prompt for the permission group (i.e., SMS in this case). The other permissions in the group (i.e., SEND_SMS and READ_SMS) are requested in the similar way. Once a permission from a permission group is requested and granted by the user, subsequent requests for other permissions from the same group are automatically granted. However, apps need to possess the individual permission in order to invoke the protected APIs.

2.3.3.2 Permission Rationale

The effectiveness of the runtime model depends on whether the permissions are requested at the right time (i.e., when they are really required by an app feature). While requesting permissions in the right context may reasonably communicate the purpose of its use, it's not always straightforward for the users who can be skeptical even if the permission is really required by the app. In such cases, users may deny the permission and get deprived of the related app feature.

To mitigate this issue, apps can show a `Dialog` or `Toast` message describing the reason why they need the permission. Android provides APIs to check whether the user has previously denied a permission: a rationale message might be useful in such cases. Apps can utilise those APIs to determine whether they should show a rationale to increase their chances to get the permission granted by the user. Even if the user has not previously denied a permission, apps can still show a rationale to justify the use.

Chapter 3

Related Work

In this chapter, we discuss related work from prior research on permission usage by Android apps in the two permission models.

3.1 Install-time Permission Model

3.1.1 Permission Usage

Prior studies identify excessive and unjustified use of permissions by Android apps. Apps are found to be transmitting sensitive user data (e.g., current location) to third parties without consent from the user [14]. Felt *et al.* [15] identify one-third of the apps in their dataset to be over-privileged having unnecessary permission usage. The outcome of these studies highlight the limitation of the install-time permission model in protecting user privacy. In the install-time permission model, users have no choice other than accepting all the permissions when installing an app, without even knowing anything about the purpose behind the usage. Kelley *et al.* [19] show that users barely understand the permission list shown to them during installation.

Gorla *et al.* [18] match Google Play Store description of the apps with requested permissions to find inconsistencies. Using their approach, they could correctly identify 56% of the malicious apps. A similar study by Watanabe *et al.* [29] identifies four major factors that contribute to the inconsistencies between app descriptions and their permission request behaviour. They find existence of third-party libraries and unspecified features as two of the major factors. These studies mainly focus on the permissions that are being requested and whether they are consistent with the app descriptions. In comparison, we consider the permissions and also their usage patterns in the apps.

3.1.2 Contextual Use of Permissions

Several studies analyse contextual resource usage in the install-time permission model. Yang *et al.* [32] define context based on environmental attributes (e.g., time of the day), to differentiate between malware and benign apps. In contrast, we define context differently, and target only regular apps. Wijesekera *et al.* [30] perform a user study to identify contextual differences and user reactions during permission use. They identify visibility to be an important context factor that validates resource access, and found user dissatisfaction while the context of permission usage change subsequently. Both these studies analyze apps developed for the install-time permission model. Therefore, it is difficult to understand what context the user might see to make a decision on the permission in the runtime model. We analyze apps that are developed for the runtime model that enables us to identify the real contexts of a request prompt.

Another study by Wijesekera *et al.* [31] combines user privacy preference and surrounding contextual cues to predict user decisions. The key idea is to differentiate between the contexts of permission use and based on prior decisions made by the user, automatically grant or request users for a permission. While identifying the contextual differences in permission usage closely relates to this work, on both occasions [30, 31], the analysis was performed on a modified version of Android protected by the old permission model with apps not designed for the runtime model. Chen *et al.* [11] propose a Permission Event Graph (PEG) model, which represents the relation between resource access and event handlers. They combine static and dynamic approaches to analyze regular and malicious apps. However, their analysis is also based on the old permission model, and cannot differentiate between the context of permission request and usage. In comparison to these studies that perform dynamic analysis to extract contextual information, we use static analysis to evaluate apps that are specifically designed and adapted for the runtime permission model.

Another line of work use the permissions listed in the manifest to generate risk signals and rank apps based on permission usage. Wang *et al.* [28] use permission request patterns to identify potentially malicious apps. Taylor *et al.* [25] develop a contextual ranking framework based on listed permissions. They propose relative ranking of apps by identifying whether an app of a specific category requests for permission(s) that are not required by other apps in the same category. However, they do not consider how the listed permissions are used by the apps. Merlo *et al.* [20] propose a risk scoring framework based on permission utilisation by apps. In comparison, we identify different contexts where the permissions are potentially used and compare them with contexts where users see a request prompt.

3.2 Runtime Permission Model

Andriotis *et al.* [6] examine the reaction of users about the runtime permission model. They find that users generally like the concept of having finer-grained control of their resources. Bonne *et al.* [10] analyse the decision of the users when they are prompted for permission in the runtime permission model. Overall, 16% permissions are denied by the users. They find that the key factor that influences user decision is the expectation about the usage of the permissions. Users make a favourable decision if they understand the need of a permission. However, even if the permission is granted for a valid purpose, it can be misused by the apps.

Votipka *et al.* [27] perform a user study to identify user reaction on permission usage in background contexts. In terms of why the permissions are accessed, users are more comfortable when the collected data is shared with the developers themselves. In terms of when a resource is accessed, users prefer resource access after direct interaction with the apps. Peruma *et al.* [23] perform another user study on the runtime permission model. They find that users generally don't feel more secured in the runtime permission model. However, compared to the old install-time permission model, users are found to be more aware about the requested permissions in the runtime permission model.

Micinski *et al.* [21] tie user interactions to resource access in the runtime permission model. They develop a dynamic analysis tool named AppTracer to analyze the extent to which user interactions and resource accesses are related. A corresponding user study reveals that users generally expect resource access right after interaction with a related app functionality. In contrast, we focus on the different Android components in which the permissions are requested and used along with user interactions.

Allen *et al.* [4] use a lightweight context-aware technique to improve malware detection. They utilise the app entry-points and sensitive APIs in their modelling and find them to be effective contextual information. They only consider the permission usage in their detection model. In comparison, we also identify request contexts and compare them with the usage contexts.

Gasparis *et al.* [17] examine the behaviour of Android apps in the runtime permission model. They analyse a set of 4,743 Android apps from different categories downloaded from Google Play Store. We analyse a much larger set of Android apps and propose ways to identify contexts that are otherwise excluded due to obfuscation. Their main focus is on developing a solution that would facilitate in-context permission request by the apps. In contrast, we compare the request/usage contexts and examine whether inconsistencies can be an indicator of maliciousness.

Gasparis *et al.* [17] also examine permission request behaviour that resembles the install-time permission model. By analysing the dataset of 2,671 apps, they find 14.07% apps request permissions when launched. Our experiment shows similar results with 13% apps in our dataset requesting permissions when first launched. However, our results are based on a much larger dataset in comparison to the 2,671 apps.

Chapter 4

Methodology

This chapter covers the definition of context, the methodology behind ContextDroid and the criteria of identifying potentially malicious behaviour based on permission request and usage contexts.

4.1 ContextDroid

We develop ContextDroid, a static analysis tool that leverages app-wide call graph and Android permission mappings to extract the contexts. The app-wide call graph is generated by using FlowDroid [7], a state of the art information flow tracking tool. We use permission mappings from Au *et al.* [8] and Backes *et al.* [9] to map API calls to associated permissions. Figure 3 shows an overview of ContextDroid.

4.1.1 FlowDroid

FlowDroid [7] is a static taint analysis tool based on Soot [26] and developed for Android. It uses Android components life-cycle to derive call-graphs of apps. Although FlowDroid is mainly developed for information flow tracking from sources to sinks, we only use its call-graph feature in ContextDroid. FlowDroid is precise, context-aware and regularly updated by the authors.

4.1.2 Permission Mapping

To identify the actual use of dangerous permissions in apps, we combine permission mappings from Au *et al.* [8] and Backes *et al.* [9]. Backes *et al.* [9] don't identify mappings for permissions that are checked by the Android OS in native code (e.g., CAMERA). Au *et al.* [8] have mappings only available for up to Android Lollipop (Android 5.1.1). Moreover, Au *et al.* [8] contains mappings for `Content Providers` (e.g., associated with permissions like READ_SMS).

Therefore, we combine the mappings from both studies to prepare a more inclusive list. In addition, we manually include some of the newer APIs from the documentation of Android that are not included in any of the mappings (e.g., Camera2 APIs associated with CAMERA permission).

4.2 Context

Our main goal is to identify usage of permissions in different situations and how they compare with scenarios where they are requested by the apps. Therefore, we rely on the Android components to define a context as they allow us to identify the differences in what the users see on screen (cf. Figure 2).

We determine the Android component being used by the users when they are prompted to grant/deny a permission and when a protected resource is accessed by the app. In particular, we identify the five Android components (i.e., `Activity`, `Service`, `Fragment`, `Broadcast Receiver` and `AsyncTask`) that trigger a resource access or permission request prompt.

4.2.1 Definition and Attributes

We consider several attributes to define a context. App Package Name (APN), represents the unique package name of the app. Permission (P) denotes a dangerous permission (e.g., CAMERA). Permission Protected API (PPA) represents a sensitive API that is protected by a permission (e.g., *getDeviceId()* API is protected by READ_PHONE_STATE permission).

A fully qualified Component Class Name (CCN) identifies the class in which a PPA is invoked (e.g., *oimmei.com.astegiudiziarie.activity.GPSQuestionActivity* is a class that invokes the *requestLocationUpdates(java.lang.String,long,float,android.location.LocationListener)* API protected by the ACCESS_FINE_LOCATION permission). Considering the fully qualified Component Class Name allows us to differentiate between classes that have the same name due to obfuscation. For example, two different classes may be renamed to the same letter by ProGuard. In that case, both classes may be considered as the same context although they are different classes or Android components. Component Type (CT) is the Android component type of the CCN (e.g., *oimmei.com.astegiudiziarie.activity.GPSQuestionActivity* is an `Activity` Component Type) .

For each App Package Name, Permission and Permission Protected API (*APN:P:PPA*), we identify the instances of the fully qualified Component Class Names and their Component Types (*CCN:CP*), each representing a different context of permission usage. Multiple instances of the same component type that use a PPA are considered as different contexts. If a sensitive resource is accessed in different `Activities`, they are considered as different contexts. If it is used in two or more `Fragments`, they are also considered as different contexts even if they reside inside the same `Activity`.

For example, *ace.astrosoft_tamil:android.permission.READ_PHONE_STATE: getDeviceId()* indicates the collection of users device ID which is protected by READ_PHONE_STATE permission. In this particular case, we determine the usage contexts by identifying all *CCN:CP* combinations.

We define a request context in the similar way discussed above. However, unlike usage contexts where each permission has its own set of protected APIs, apps only need to use a small set of APIs for requesting any dangerous permissions. Therefore, the only difference with identifying usage contexts is that for request contexts, we do not need to consider associated APIs. To define contexts where a permission is requested, we identify all combinations of *CCN:CP* for a given *APN:P*.

`Activity` and `Fragment` are considered as foreground contexts as the user can directly interact with them through the UI. `Service`, `Broadcast Receiver`, and `AsyncTask` do not have UI and we consider them to be background contexts.

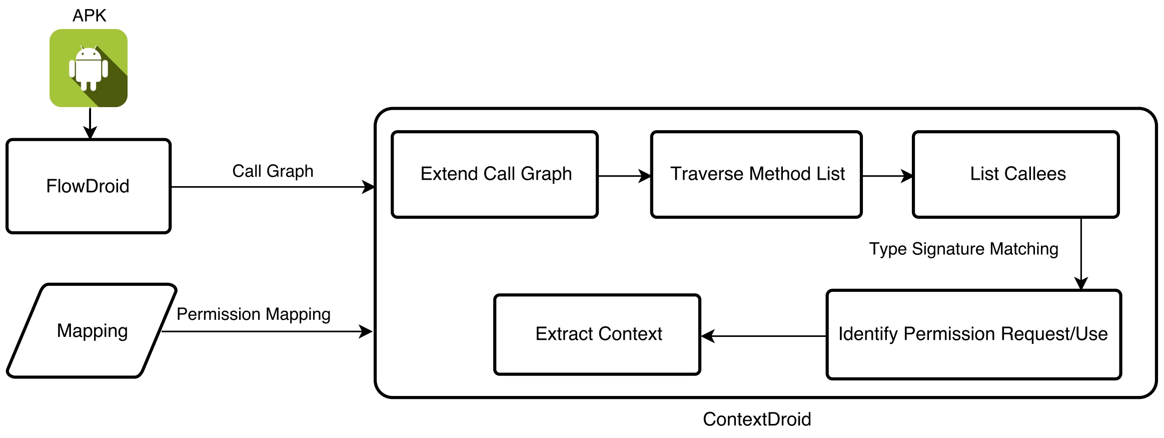


Figure 3: Overview of ContextDroid

Our definition does not cover the actual functionalities attached to these components. However, using the same permission and API in different components indicates that it is used across multiple functionalities within the app where some of them might be unexpected, unexplained or malicious.

4.2.2 Context Detection

We use the following information to determine the contextual information: *Activation event* (an entry point of the call graph), *Request API* (used to show permission prompts at runtime) and *Permission Protected API* (PPA).

4.2.2.1 Usage Context

We search each method in the call graph and identify calls to PPAs. We match the type signature of the APIs inside the method with PPAs from the permission mappings. If we find a match, we first check whether it is a standalone method representing an *activation event*. If not, we traverse back to all the callers of that method until we find the activation events. We then extract contextual attributes that include the App Package Name (APN), Permission (P), Permission Protected API (PPA), Component Class Name (CCN) and Component Type (CT).

To identify the CT, we first check the name of the component and its parent class. If they match with any of the components (e.g., `Activity`), we store the CCN and CT. For example, if we find an activation event in LoginActivity (a subclass of `Activity`), we identify it as an `Activity` component.

Another way to identify the Android component type is to check the Android-Manifest file that lists the `Activities`, `Services` and `Broadcast Receivers` of an app. However, PPA calls from `Fragment` and `AsyncTask` would not be identified as they are not included in the AndroidManifest file.

To handle class name obfuscation where component type cannot be identified from the class name or its parent class, we recursively traverse back to the parent classes of that method and identify whether it is a child class of any of the Android component classes. For example, we can not identify the CT from an obfuscated CCN such as *a.b.c.d.ef*. In this case, we iterate through its parent classes to identify the CT.

4.2.2.2 Request Context

Apps can request permissions by using one of the Request APIs. To infer the requesting context, we identify calls to different instances of *requestPermissions()* APIs and follow a similar approach described for permission usage to identify the relevant contextual information.

However, unlike the system API calls that cannot be obfuscated by ProGuard (with the Google Play Services library being an exception), support library APIs that are shipped with the APK can be obfuscated in the release build (unless the developer excludes certain classes from obfuscation). Support library contains APIs that can be used to request permissions. For example, apps can request a permission by calling the *requestPermissions()* API from the *ActivityCompat* or *ContextCompat* classes (both available in the support libraries).

To handle such instances, we use partial type signature matching to identify the context of permission request. We first identify whether the method contains permission strings, e.g., android.permission.AUDIO. If found, we examine subsequent API calls that take the permission strings as a parameter. Specifically, we identify whether the package name of the method partially matches with a support library package (e.g., *android.support.v4.a.a* partially matches with the package name of support library version 4). If a match is found, we further compare the parameter signature of the API with request APIs from the support library. If the partial type signatures match, we consider this as an instance where permission is requested.

4.2.3 Extended Call Graph

Identifying obfuscated `Fragments` (e.g., by ProGuard [12] or other similar tools) that are derived from support libraries is not straight forward. To identify `Fragment` contexts which are otherwise excluded from the call graph (e.g., due to obfuscation), we propose an extended call graph in ContextDroid.

We first iterate through all the methods in the call graph and identify the class in which they are declared. If the component type of the class cannot be identified, we iterate through the parent classes to determine whether they can be identified as a subclass of an Android component. If the component type cannot be determined, we attempt to find whether the class is derived from support library `Fragment`.

We start with the package name of the method and perform partial matching with support library package and iterate through the parent classes and their package names until we find a match. If a match is found, we extract the method list of that class.

Android `Fragments` used in apps must override the *OnCreateView()* method. To determine whether the class is a `Fragment` component, we further match the return and parameter types (defined as the sub-signature) of the listed methods with *OnCreateView()*. If the sub-signature matches, we tag it as a `Fragment` and include the methods of that class in the call graph.

4.3 Contextual Differences

We analyze the differences in terms how apps request and use permissions in multiple contexts. Following the best practices [2] and developer policy for permissions [3] from Google, we flag apps for potentially using permissions in unexpected contexts.

Our mechanism, for the most part, is based on two situations:

- Apps using dangerous permissions in dissimilar and disconnected contexts (*Dissimilar Usage Contexts*).
- Apps using dangerous permissions excessively in multiple contexts including third-party libraries (*Multiple Usage Contexts*).

In this section, we describe our mechanism for flagging apps.

4.3.1 Dissimilar Usage Contexts

We first check if both request and usage contexts of an app are identified by ContextDroid. If only the usage contexts are identified, we follow the flagging mechanism for multiple usage contexts; cf. Section 4.3.2.

We iterate through all the permissions with identified usage contexts in the app. For each permission, we search for the request contexts. If found, we check whether the number of request contexts is equal to the number of usage contexts.

If the number of contexts are equal, we determine whether all the request and usage contexts match. We compare whether for each APN:P:CCN:CP combination of usage contexts there exists a request context with the exact same combination.

For instance, we find one usage context of ACCESS_COARSE_LOCATION permission (P) in the *catholic.bible.download* (APN) app inside an `Activity` component (CT) named *catholic.bible.download.MainActivity* (CCN). In this case, we examine whether there exists a request context with exact same combination of the contextual attributes. This comparison makes sure that the permissions are used in a context where the user is likely to see a permission request prompt to make an informed decision.

Even if all the contexts match, it is possible that the user would grant access to the app but not the third-party library. However, once the app is granted a permission, the attached third-party library can also use it. In other words, if a permission is granted based on an app context, it can be used in a context from an attached third-party library. Therefore, if all the contexts match for a particular permission, we examine whether there exists a request context within the app and a usage context in a third-party library. If found, we flag the app as potentially using a permission for unwanted purpose.

If the contexts don't match, we check whether there exists a connection between them. We check if there's any background `Service` usage context for a permission. For each `Service` context, we check the components that can start the `Service`. If any of the services are invoked from a component where there is no request prompt, we flag the app. We note that this criterion doesn't apply in case of READ_SMS permission. We do not allow the usage of READ_SMS permission in a `Service` because reading all inbox messages in the background has rare use cases and can be violated (see e.g., Section 5.4). In fact, Google recently prohibited the use of

READ_SMS in any app except the ones with capabilities to become the default SMS app [3].

Furthermore, we flag an app if there is a difference between the number of usage contexts and request contexts. This difference simply means the permission is used in more contexts than where it is requested. In other words, if a permission is granted in one context, it is used in other contexts where the user may not be aware.

4.3.2 Multiple Usage Contexts

For a particular permission, if ContextDroid could identify only the usage contexts, we flag the app based on the number of usage contexts and existence of third-party libraries. We flag the app if a permission is used in multiple (i.e., 2 or more) contexts and at least one of them includes a third-party library. We follow a similar approach when flagging apps for which we could identify only the usage contexts.

However, determining the number of contexts that should be considered as valid is non-trivial as it simply depends on the functionalities provided by the app. Depending on the app functionality, using permissions in multiple contexts can be valid. In our analysis, we allow only 1 valid usage context. In reality, this number should vary based on the app functionality. For example, a messaging app using SMS permission in multiple different contexts should ideally be valid. Therefore, while analysing such apps, number of valid context(s) should be increased before flagging apps for violation.

4.4 Foreground-Background Component Relationship

When a permission is granted by the user in a foreground component, the protected resource might be accessed in a background `Service`. Although `Services` are listed in the AndroidManifest, they need to be instantiated by another component (e.g., `Fragment`). If the two components are connected (e.g., `Service` initiated by an `Activity`) and the resource is accessed only in that `Service`, it's more likely to be a legitimate resource access (except READ_SMS), provided the user granted the permission based on what feature is being used. Therefore, while analysing contextual differences in resource access, we need to consider if there is any relation between the foreground and background components.

To identify such relationships, we first list the `Services` of an app from its AndroidManifest file. While traversing the call-graph methods, we identify whether any of the listed `Services` were instantiated in them. If identified, we check whether there exists a call to the *startService(android.content.Intent)* API that indicates the `Service` can be started from that method. We then list the class name (CCN) and component type (CT) of that method in our database.

4.5 Permission Rationale

Android provides the *shouldShowRequestPermissionRationale()* API that returns whether an app should show the rationale behind the usage of permissions. The API returns true if the permission was previously requested but denied indicating that the purpose of the permission was not effectively communicated to the user.

We determine whether apps show any rationale for the requested permissions by identifying calls to the *shouldShowRequestPermissionRationale()*, *show()* and *makeToast()* APIs. If we identify a request context in a method, we check whether it contains calls to the *shouldShowRequestPermissionRationale()* API. If found, we check the method to find invocation of the *show()* and *makeToast()* APIs. These two APIs are used to invoke a pop-up Dialog or a Toast message.

If none of these APIs are found, we retrieve the callees of that method and search them to identify the API calls. If not found, we conclude that the app doesn't show any rationale to the user for the permissions.

4.6 Third-Party Library Detection

Android apps often use third-party libraries for different reasons (e.g., for showing ads). It is possible that a permission requested by an app for legitimate use can be utilised for unknown/malicious purposes by attached third-party libraries. Therefore, we need to identify whether a permission is requested or used in a context from a third-party library.

We determine this by identifying whether a part of the APN of an app is a sub-string of the CCN of a context. Android follows Java package naming convention and the first two parts of the package name usually includes the company name [22]. Therefore, we first tokenize the APN by using the (.) delimiter and store the first two parts of it. For example, from APN *academy.itmons.elderigok* we take *academy.itmons*. For each request and usage contexts found for *academy.itmons.elderigok* app, we identify whether the CCN of the contexts contain *academy.itmons*. If not found, we conclude that the context is from a third-party library.

4.7 Comparison with the Install-time Permission Model

The major difference between the install-time model and the runtime model is that not all the permissions are granted by default if the app is installed in the user device. However, if apps request all their required dangerous permissions when they are first launched, it becomes similar to the install-time model and the whole point of in-context permission request model (i.e., runtime model) fails.

We want to identify whether apps are still following the install-time model by simply requesting all permissions at once during launch. To identify such phenomenon, we first extract the `LAUNCHER Activity` from the AndroidManifest. For each app, we then see whether all the dangerous permissions are requested in that `Activity`. If all of them are requested in the same `LAUNCHER Activity`, we consider that app as showing behaviours similar to the install-time permission model.

4.8 Comparison with VirusTotal Detection

Detecting malware is not our primary goal. However, we want to see whether our app flagging mechanism based on contextual use of permissions can be linked to malicious behaviour.

VirusTotal combines the results of various Anti-Virus (AV) tools for software and applications. The result consists of the number of tools that identify the APK as malicious. We compare the apps flagged by our mechanism with VirusTotal by identifying whether the flagged apps were detected as malicious by any of the AVs.

Chapter 5

Results

In this chapter, we discuss our dataset, experimental setup and the results.

5.1 Dataset and Setup

In this section, we describe our dataset selection criteria and the performance of our analysis.

5.1.1 Dataset Collection

5.1.1.1 Selection Criteria

We want to analyse how the dangerous permissions are requested and used. The runtime permission request was introduced in Android API version 23. We primarily select apps that target the runtime permission model (e.g., targetSdkVersion = 23). However, not all the apps that are developed for the runtime-permission model need dangerous permissions. Therefore, we select apps that declare dangerous permissions in their AndroidManifest.

5.1.1.2 Dataset

We collect our apps from the AndroZoo project [5]. Our dataset comprises 62,340 different versions (APKs) of 42,940 apps from 48 categories. New versions of apps are often released with new functionalities that may require accessing a permission protected resource [24]. Therefore, unless otherwise specified, we consider these

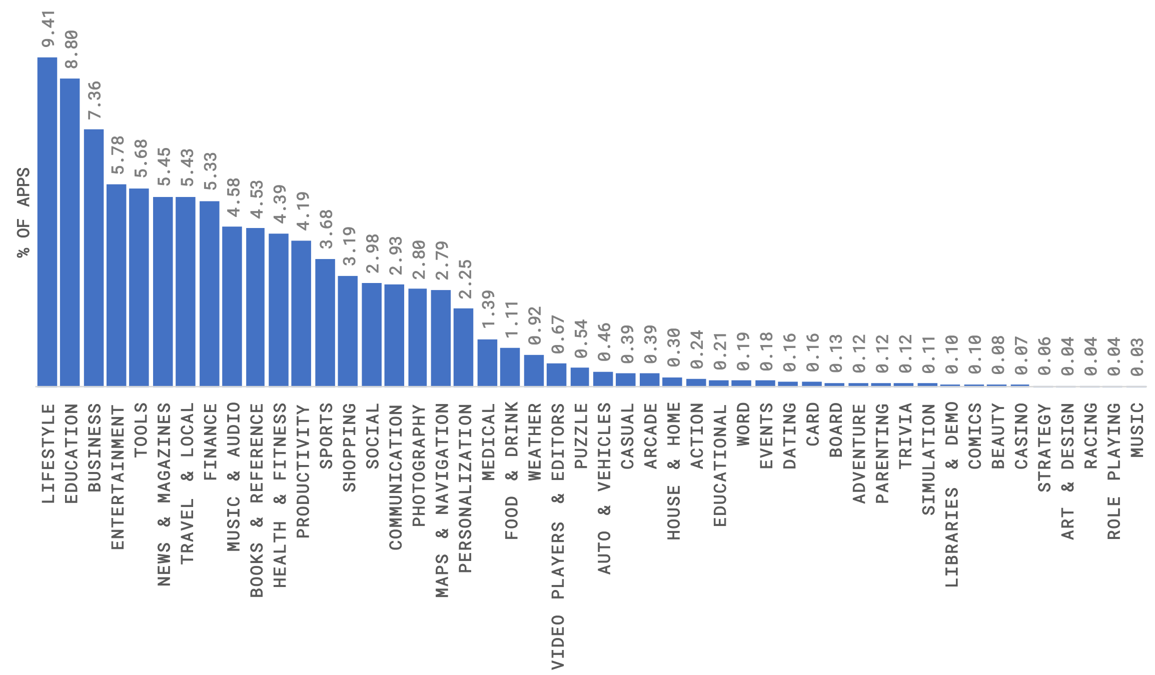


Figure 4: Category of apps in our dataset collected from the AndroZoo [5] project

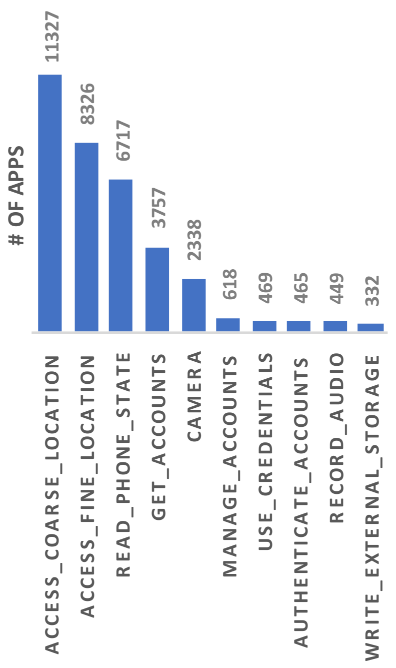
APKs as different apps in our analysis. Figure 4 shows the percentage of apps from different categories.

5.1.2 Performance

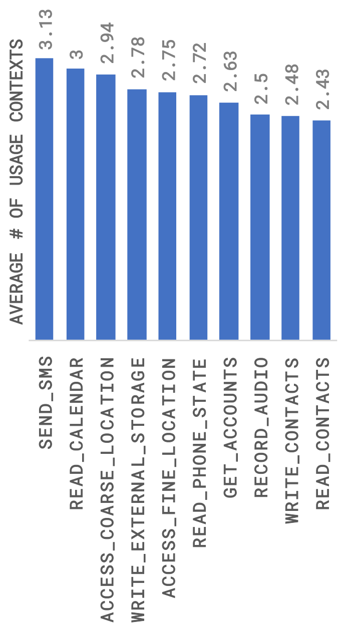
We perform our analysis on an Intel Core i7 3.60GHz processor with 24GB of memory running Ubuntu 16.04. For each app, our analysis takes on average 43.42 seconds to identify the contexts, including time taken by FlowDroid. Note that we first use FlowDroid to generate an app-wide call graph that takes on average 42.40 seconds for a regular app (of size around 6MB).

5.2 Permission Request and Usage In Multiple Contexts

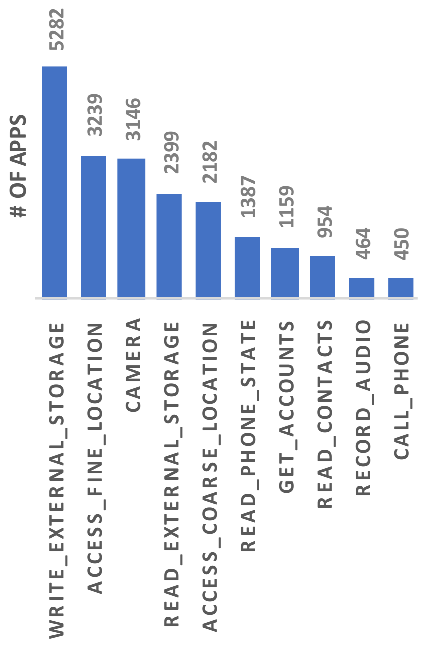
We find that ACCESS_FINE_LOCATION (29%), ACCESS_COARSE_LOCATION (21%) and READ_PHONE_STATE (17%) permissions are used by apps in multiple



(a) Permission usage in multiple contexts



(b) Average number of usage contexts



(c) Permission requests in multiple contexts

Figure 5: Permission requests and usage in multiple contexts by apps and the average number of usage contexts for different permissions

contexts more frequently compared to others. These permissions allow apps to collect sensitive information such as users current location and device information. Figure 5a shows the permissions that are used by the apps in multiple contexts.

Although not used by many apps, SEND_SMS and READ_CALENDAR permissions have higher average number of usage contexts compared to others. Apps that use these permissions in multiple contexts have on average three or more contexts. Figure 5b illustrates the average number of contexts for the permissions that are used in multiple contexts.

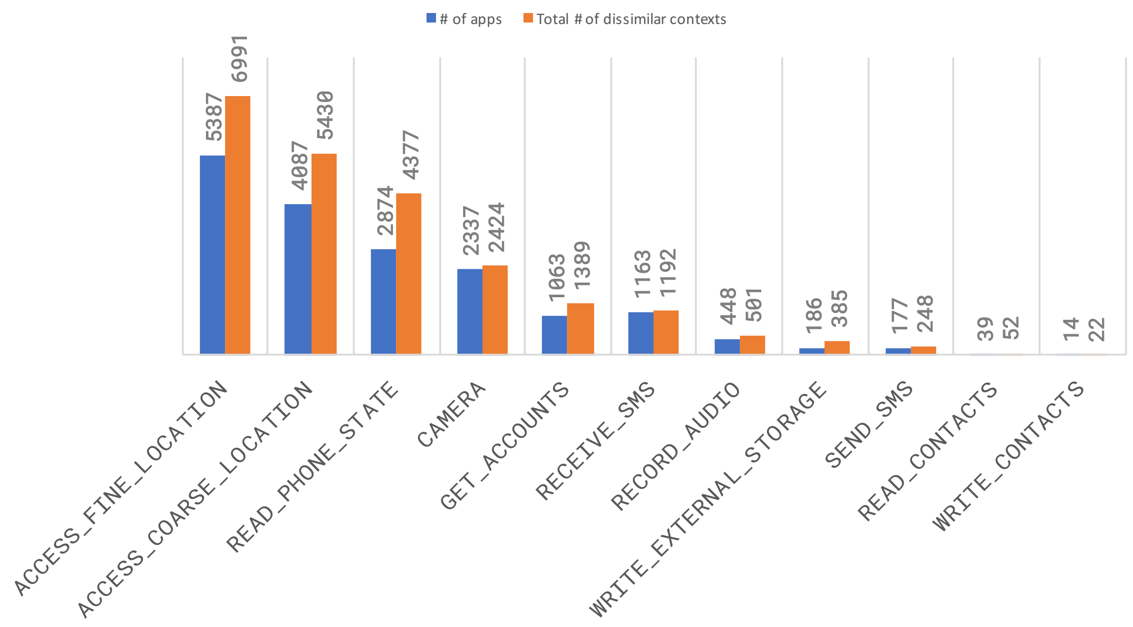


Figure 6: Permissions that are used in dissimilar usage contexts by apps

We find that WRITE_EXTERNEL_STORAGE, ACCESS_FINE_LOCATION, CAMERA, READ_EXTERNEL_STORAGE and ACCESS_COARSE_LOCATION permissions are requested in multiple contexts more frequently by apps, suggesting they are more context aware while requesting for these permissions. Figure 5c gives an overview of the number of apps that request permissions in multiple contexts.

We find request and usage contexts of permissions in 22,567 apps. 7,753 apps (34.36%) use at least one permission in contexts that don't match with the contexts of where they are requested.

Figure 6 demonstrates the number of apps that use permissions in dissimilar contexts. Compared to others, CAMERA, RECORD_AUDIO and RECEIVE_SMS are used less often in dissimilar contexts as these permissions on average have one context where there is no permission prompt. As shown in Figure 5c, WRITE_EXTERNEL_STORAGE is requested in multiple contexts the most suggesting apps are more careful when requesting this permission. This is understandable because this permission allows them to write and access the most privacy sensitive files (e.g., personal photos) in the user device. There are quite a few apps that still use this permission in dissimilar contexts (cf. Figure 6), though the number can be considered negligible compared to others.

Other permissions on average have more than 1.3 dissimilar contexts. ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, READ_PHONE_STATE and GET_ACCOUNTS are the most abused in that regard. We find apps to be less context aware while requesting these permissions.

5.3 Contextual Differences

In this section, we discuss the results based on our app flagging mechanism (see Section 4.3) and compare them with VirusTotal detection.

5.3.1 Flagged Apps

We first discuss the apps flagged for *Dissimilar Usage Contexts* and *Multiple Usage Contexts* separately and then discuss the overall results. As discussed in the previous section, we identify request and usage contexts in 22,567 apps. Therefore, the percentages of *Dissimilar Usage Contexts* are based on those 22,567 apps. We find usage contexts in 38,838 apps. Our discussion on the *Multiple Usage Contexts* and the overall results are based on 38,838 apps.

VirusTotal Detection. Our dataset contains 6,295 versions (APKs) of 5,499 apps that are detected as Potentially Unwanted Programs (PUP) or Malware by at least one Antvirus engine in VirusTotal. It is possible that different versions of the same app with similar features are detected by VirusTotal and that would make our results somewhat biased. However, we find that not all versions of the same app are detected by VirusTotal engines.

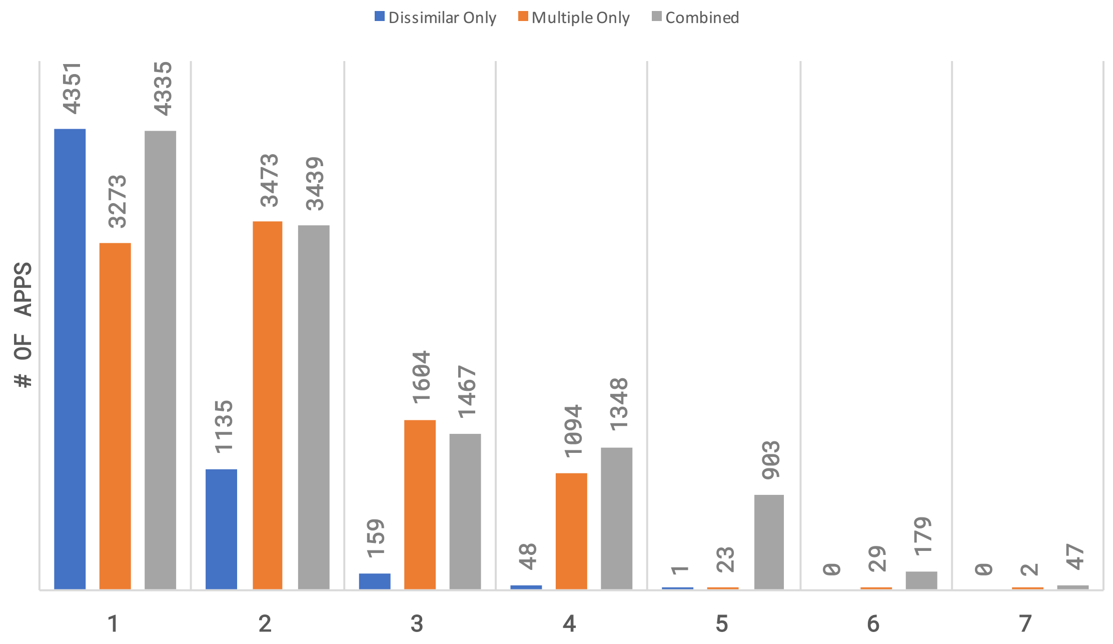


Figure 7: Apps flagged for multiple and dissimilar usage contexts

5.3.1.1 Dissimilar Usage Contexts

We flag 5,694 (25% of the 22,567 apps) apps for dissimilar usage contexts. 4,351 apps have only one dissimilar context while 1,135 apps are flagged for having two dissimilar contexts. 208 apps are flagged three or more times by our mechanism. Figure 7 shows the number of apps and how many times they are flagged.

Comparison with VirusTotal. 2,378 apps (i.e., 37% of the 6,295 VirusTotal detected apps and 41% of the 5,694 flagged apps) flagged for dissimilar usage contexts are also detected as malicious by at least one VirusTotal engine.

5.3.1.2 Multiple Usage Contexts

We flag 9,498 (24% of the 38,838 apps) apps for (multiple) usage contexts in the app and third-party library. 3,273 apps are flagged only once while 6,225 apps are flagged multiple times. These apps use more than one permission in multiple and third-party contexts. Figure 7 shows the summary of apps that use permissions in multiple contexts and the number of times they are flagged.

Comparison with VirusTotal. 3,934 apps (i.e., 62% of the 6,295 VirusTotal detected apps and 41% of the 9,498 flagged apps) flagged for multiple usage contexts are also detected as malicious by at least one VirusTotal engine.

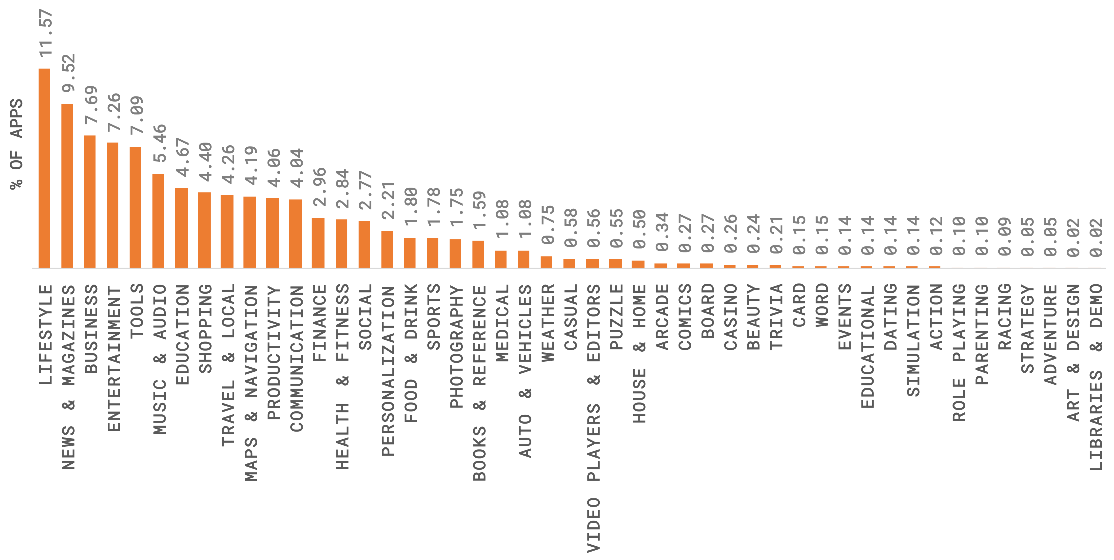


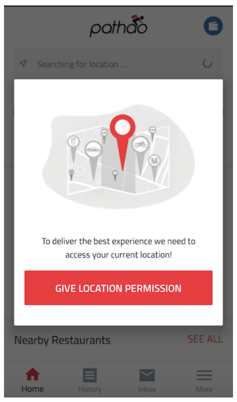
Figure 8: Category of the apps flagged for multiple and dissimilar usage contexts

5.3.1.3 Overall Comparison

Overall, by combining the two criteria discussed above, we flag 11,728 (30.20%) apps. 4,335 apps are flagged only once. The rest of the apps are flagged multiple times. Figure 7 illustrates the number of times these apps have used a permission in multiple or dissimilar contexts. Figure 8 shows the category of the flagged apps.

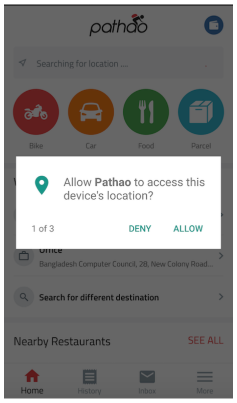
Comparison with VirusTotal. 4,073 apps (i.e., 64.70% of the 6,295 VirusTotal detected apps and 34.72% of the 11,728 flagged apps) flagged for multiple usage contexts are also detected as malicious by at least one VirusTotal engine.

Our dataset contained 3,281 apps that are detected by at least two VirusTotal engines and our mechanism flagged 2694 (82.10% of the 3281 apps) of them. if we consider detection by more than five VirusTotal engines, our dataset contained 840 apps and our mechanism flagged 692 (82.38% of the 840 apps) of them. Furthermore, 341 apps in our dataset are detected as malicious by more than 10 VirusTotal engines. Our flagged apps included 290 (85.04% of the 341 apps) of them.



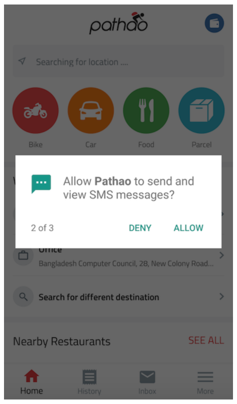
To deliver the best experience we need to access your current location!

(a) Rationale for Location access



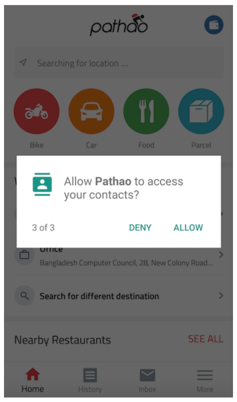
Allow **Pathao** to access this device's location?

(b) Location Request Dialog



Allow **Pathao** to send and view SMS messages?

(c) SMS Request Dialog



Allow **Pathao** to access your contacts?

(d) Contacts Request Dialog

Figure 9: Ride-sharing app Pathao showing permission rationale for accessing device location. In addition to requesting location access (9b), it also requests for SMS (9c) and contacts permission (9d)

5.3.2 App Versions

Our dataset contains 18,318 versions of 6,215 apps. We compare different versions of each of these apps in terms of the number of times they are flagged. We find that contextual use of permissions vary across different versions. 856 apps (14% of the 6,215 apps) have contrasting flag count across different versions (some versions of the apps are flagged while others aren't).

5.4 Case Study 1

We discuss a ride-sharing app named Pathao as a case study. Pathao is the most popular ride-sharing platform in Bangladesh. Recently, Pathao has been subjected to widespread criticism for collecting SMS messages and contacts from Android devices. Analysis of the network traffic reveals that the collected data was uploaded to their own server [13].

The most recent version of Pathao removed the code used for collecting the data. However, the fact that they were able to collect this information (without mentioning anything in their privacy policy) while still being available in the Google Play Store is surprising enough.

Therefore, we want to follow our mechanism while discussing Pathao to show that it can be useful to identify potentially malicious behaviour in regular apps early in the vetting process and subject them to thorough analysis before they are available in the Google Play Store.

As discussed in Section 2.3.1, Android does not show the individual permissions that are requested by apps. Instead, it shows request for the permission group. Therefore, we discuss permission requests in Pathao in terms of permission groups.

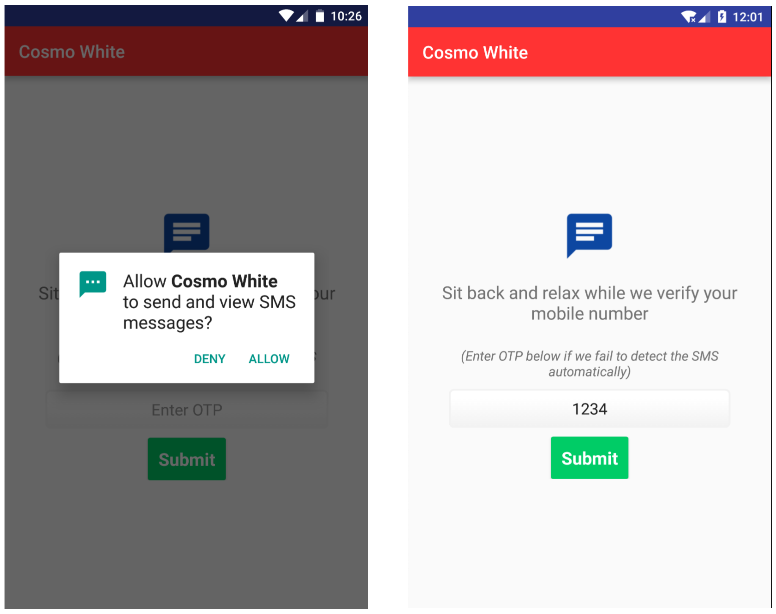
Pathao shows a rationale in a `Fragment` context for accessing device location as soon as the users log in (see Figure 9a). Although we couldn't find usage of location (due to the Play Services Library), Figure 9a suggests it's necessary for the purpose of the app (e.g., location based features). However, in addition to requesting location access (Figure 9b), it also requests for SMS (Figure 9c) and contacts (Figure 9d) access. There are two other contexts in different `Activities`, where we find requests for SMS.

We find one context for RECEIVE_SMS usage in a third-party library (i.e., Facebook account kit). READ_SMS and READ_CONTACT are used in `AsyncTask` contexts inside a `Service`. The contexts of READ_SMS, RECEIVE_SMS and READ_CONTACT usage do not match the contexts in which they are requested. Overall, Pathao is flagged 3 times in our analysis as it requests and uses READ_SMS, RECEIVE_SMS and READ_CONTACT permissions in dissimilar contexts.

Once flagged by our mechanism, we go for a manual inspection of the permission usage. We first go through the source code to see what the permissions are being used for. In case of RECEIVE_SMS, we conclude the usage as legitimate as account kit is used for One Time Password (OTP) verification. Next, we move on to READ_SMS and READ_CONTACT. We find that Pathao retrieves the complete list of the SMSs and contacts from the device. Analysing its network traffic reveals that the information is uploaded right away after the permissions are granted.

Interestingly, Pathao justified their actions by attributing the need of SMS permission to OTP verification. Even if it is true, the app still used the permission for a different purpose (different context in our methodology) in a malicious way. However, as our analysis reveals, the OTP is performed by Facebook account kit that doesn't need the SMS permission to validate the user unless the Google Play Services is not installed in the device [1].

We believe that Pathao is a strong case of permission abuse in dissimilar contexts.



Allow **Cosmo White** to send and view SMS messages?

Sit back and relax while we verify your mobile number

(Enter OTP below if we fail to detect the SMS automatically)

Figure 10: Cosmo White app requesting SMS permission for OTP verification

It uses the granted permissions in unexpected/malicious contexts, while still being active as a regular app in the Google Play Store. Even VirusTotal considers this app as benign whereas our mechanism flagged it for potential violation. This procedure can be followed for other apps early in their vetting process before they are made available in the Google Play Store.

5.5 Case Study 2

We discuss another case study of how our flagging mechanism can be useful in identifying seemingly benign apps that use permissions maliciously. We discuss a personal finance app named Cosmo White.

Cosmo White got flagged for two dissimilar usage contexts. Static analysis reveals that SMS permission is requested in an `Activity` context and used in a background `Service` context. We then manually inspect the app to determine the purpose the SMS permission. Figure 10 shows the permission request prompt for the SMS permission when the app is launched. It seems the SMS permission is really required from the given context of the app (i.e., OTP verification).

However, manual inspection of the code reveals that the OTP verification is not performed by SMS at all. We find that when the SMS permission is granted, the app starts a background `Service` that collects all the messages from the device and then sends them to their server. The OTP verification process succeeds as soon as the SMSs are uploaded to the server. Cosmo White contains an input field where the users can manually input their OTP verification code (if they don't want to grant the SMS permission). However, the decompiled source code reveals that the value of this input field is never used for OTP. The user given input will never work unless it is '1234' (hard-coded in the app).

5.6 Install-time Behaviour

We find a number of apps that request all the permissions during app launch with 13% apps having permission prompts in their `LAUNCHER Activity`. These apps show the typical permission request behaviour of the install-time model.

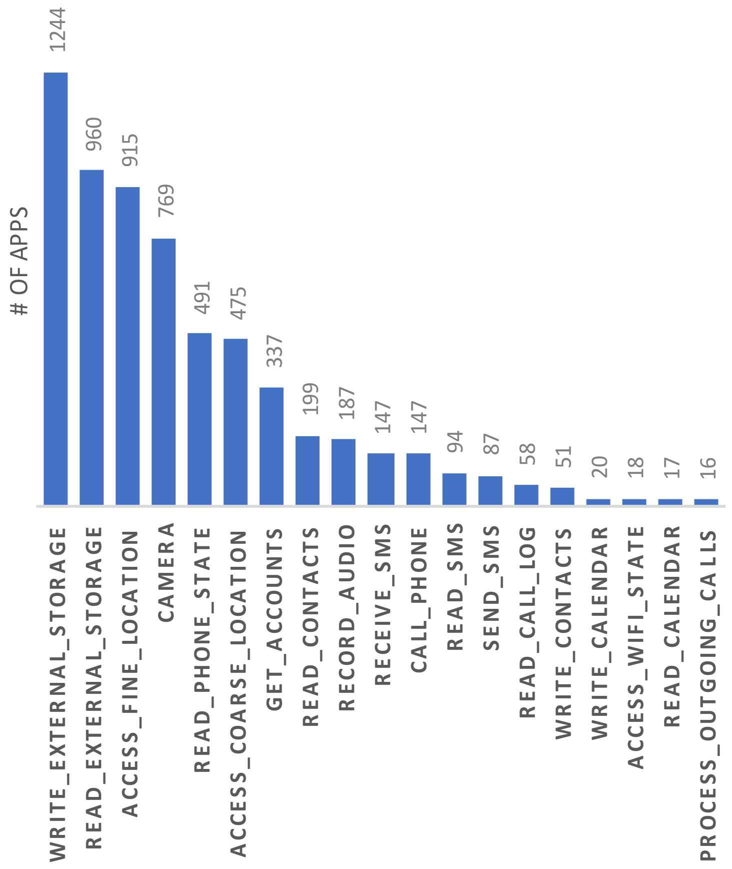


Figure 11: Rationale displayed by apps for permission requests

If we consider requesting all the permissions in one context - not necessarily in the `LAUNCHER Activity` alone, 15% apps request the required permissions in the same context. Although, not exactly behaving like an app targeted for the install-time model, these app simply ask for all permissions at once.

5.7 Permission Rationale

Only 5% apps in our dataset show rationale before requesting a permission (if the user previously denied it). Figure 11 shows the permissions for which apps show rationales. Apps show rationales for READ_EXTERNEL_STORAGE, WRITE_EXTERNEL_STORAGE ACCESS_FINE_LOCATION and CAMERA permissions the most.

As previously noted (see Section 5.2) in the discussion of multiple request contexts and dissimilar contexts, apps show rationales for storage permissions the most. Although, the overall percentage of rationales is not noteworthy, evidently, apps are more conscious about accessing the storage and camera than any other resources.

Chapter 6

Discussion and Future Work

6.1 Discussion

This thesis presents the first large-scale study of the contextual differences in Android apps in the runtime permission model. We present ContextDroid, a static analysis tool that extracts the contexts in which apps request and use dangerous permissions. Our findings suggest that the apps are far from being context aware while requesting for the dangerous permissions at the right time.

Permissions are used in various contexts where there are no permission prompts shown. We identify dissimilarity in the usage contexts in 34.36% of the apps. While such dissimilarity can be a attributed to various non-malicious reasons such as developer negligence or app functionality, we find that they can also be linked to malicious behaviour.

Our analysis shows that apps are more context aware about specific resources than others. Overall, we find that Storage and Camera resources are handled with caution in terms of contextual use by the apps. The ratio of dissimilar contexts for these permissions are lower than that of others. This is also complemented by the fact that apps show more rationales for these permissions.

In our flagging mechanism, we do not classify the apps as malware, as they were - at some point in time, uploaded to the Google Play Store and survived there (at least for a period of time). Our primary focus is to identify apps that are potentially using permissions in unexpected contexts in the runtime-model. It is possible and in fact, quite prevalent that a permission can be legitimately used across different functionalities within an app.

However, we find that using permissions in many and dissimilar contexts can be connected to maliciousness to an extent. Our app flagging mechanism that is simply based on the best practices of permission request and usage in the runtime model shows promise. Comparing our results with the detection rate of VirusTotal shows that using permissions in multiple and dissimilar context can indeed be malicious. Apps flagged by our methodology overlap with 64.70% of the VirusTotal detected apps.

We identify apps that behave in a similar fashion compared to the old install-time model. 13% apps request all the dangerous permissions that they need when they are first launched. The percentage of apps showing rationale explaining the purpose of the permission usage is also negligible. Apart from a few (e.g., WRITE_EXTERNEL_STORAGE), the reason behind the usage of permissions is not communicated to the user even if the user denied the request previously.

6.2 Limitations and Future Work

When defining context, we do not consider the environmental attributes (e.g., device is locked or not). Previous research identify apps that use such attributes to trigger a behaviour to be outright malware [32]. As our focus is on regular apps and to identify whether runtime-model has changed the way permissions are used, we mainly consider the type of component and their relationship (e.g., Foreground - Background) to define unexpected contexts.

In this thesis, we identified foreground-background component relationship. ContextDroid can be further extended to establish relationship between similar type of components (e.g., Activity-Activity) to get finer-grained insight and we leave it as a future work.

Our definition of context does not take into account the apps' feature. Our approach is based on quantity and similarity rather than necessity. To infer what the users might consider as unexpected behaviour is non-trivial and varies greatly depending on the how they view the importance of their privacy [30]. While a finer-grained approach can be taken to further differentiate the context, we believe that our definition provides an overall view of how permissions are used in various components (contexts in our definition) by the apps.

We only identify permission rationale in methods or callees of the methods where a permission is requested and the *shouldShowRequestPermissionRationale()* API is called. While this ensures that we identify whether apps show any rationale in case of a previous rejection by the users, it does not cover all the scenarios. For example, it is possible that the app shows a rationale outside of this workflow, even before requesting the permissions. In such cases, our methodology does not identify whether a rationale is shown. Besides, we do not consider what kind of rationale is shown for a particular permission. Identifying the meaning of the rationale message would need natural language processing that is out of the scope of this thesis.

We could not identify request and usage contexts of all the apps in our dataset. Furthermore, we could not identify the contexts for all the permissions in an app. It is non-trivial to identify at scale whether the permission usage contexts are not identified or the permissions are not used at all. Apps often specify permissions in the AndroidManifest without actually using them [20].

We randomly select 25 apps for which we could not identify the request or usage contexts. We identify several reasons after manually inspecting the apps. We find factors such as unused permissions listed in the manifest, use of reflection based libraries to request/use permissions, use of permissions in the native code, analysis timeout or exception occurred during analysing broken or incompatible (e.g., with FlowDroid) APKs. Moreover, we find instances of third-party libraries integrated with the apps that only check whether a permission is granted or not without explicitly requesting that permission. These libraries use a permission only if the host app has been granted the permission.

In addition, ContextDroid cannot identify the APIs that are protected by dangerous permissions and the APIs for permission requests inside methods that are not included in the call graph. To address this limitation, we propose an extended call graph that identifies obfuscated `Fragment` and API. However, we do not consider advanced forms of obfuscation and leave it as future work. We identify sensitive API calls based on permission mappings from prior work. If an API is missing in the mapping list, ContextDroid will fail to identify the usage of the associated permission.

ContextDroid can be extended to work in collaboration with a dynamic analysis tool to overcome the limitations of not identifying all request and usage contexts.

ContextDroid's output can be used to trigger the specific components where there is a permission prompt shown or a resource is accessed. By using this targeted dynamic execution, ContextDroid's output can be validated and if there are any requests or usage that could not be previously identified, can be found. The combined list from the static and dynamic analysis would provide a comprehensive list of contexts.

Bibliography

[1] Android project configuration - account kit - documentation. `https://developers.facebook.com/docs/accountkit/android/configuration/`.

[2] App permissions best practices — Android developers. `https://developer.android.com/training/permissions/usage-notes/`.

[3] Permissions — privacy, security, and deception - developer policy center. `https://play.google.com/about/privacy-security-deception/permissions/`.

[4] Joey Allen, Matthew Landen, Sanya Chaba, Yang Ji, Simon Pak Ho Chung, and Wenke Lee. Improving accuracy of Android malware detection with lightweight contextual awareness. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 210–221. ACM, 2018.

[5] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: collecting millions of Android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 468–471. ACM, 2016.

[6] Panagiotis Andriotis, Martina Angela Sasse, and Gianluca Stringhini. Permissions snapshots: Assessing users' adaptation to the Android runtime permission model. In *Information Forensics and Security (WIFS), 2016 IEEE International Workshop on*, pages 1–6. IEEE, 2016.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[8] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: analyzing the Android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[9] Michael Backes, Sven Bugiel, Erik Derr, Patrick D McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the Android application framework: Re-visiting Android permission specification analysis. In *USENIX Security Symposium*, pages 1101–1118, 2016.

[10] Bram Bonné, Sai Teja Peddinti, Igor Bilogrevic, and Nina Taft. Exploring decision making with Android's runtime permission dialogs using in-context surveys. USENIX Association, 2017.

[11] Kevin Zhijie Chen, Noah M Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in Android applications with permission event graphs. In *NDSS*, page 234, 2013.

[12] Android Developers. Shrink your code and resources. `https://developer.android.com/studio/build/shrink-code.html/`.

[13] Ishtiaque Emon. Dissecting pathao - an uber-like app that steals sms, contacts, app list and more., Nov 2018.

[14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[15] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[16] Adrienne Porter Felt, Serge Egelman, and David Wagner. I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns. In *Proceedings*

of the second ACM workshop on Security and privacy in smartphones and mobile devices, pages 33–44. ACM, 2012.

[17] Ioannis Gasparis, Azeem Aqil, Zhiyun Qian, Chengyu Song, Srikanth V Krishnamurthy, Rajiv Gupta, and Edward Colbert. Droid M+: developer support for imbibing Android's new permission model. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 765–776. ACM, 2018.

[18] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.

[19] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. A conundrum of permissions: installing applications on an Android smartphone. In *International Conference on Financial Cryptography and Data Security*, pages 68–79. Springer, 2012.

[20] Alessio Merlo and Gabriel Claudiu Georgiu. RiskInDroid: machine learning-based risk analysis on Android. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 538–552. Springer, 2017.

[21] Kristopher Micinski, Daniel Votipka, Rock Stevens, Nikolaos Kofinas, Michelle L Mazurek, and Jeffrey S Foster. User interactions and permission use on Android. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 362–373. ACM, 2017.

[22] Oracle. Naming a package. `https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html/`.

[23] Anthony Peruma, Jeffrey Palmerino, and Daniel E. Krutz. Investigating user perception and comprehension of Android permission models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, pages 56–66, New York, NY, USA, 2018.

[24] Jingjing Ren, Martina Lindorfer, Daniel Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. Bug fixes, improvements,... and privacy leaks– a longitudinal study of pii leaks across Android app versions. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[25] Vincent F Taylor and Ivan Martinovic. Securank: Starving permission-hungry apps using contextual permission analysis. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 43–52. ACM, 2016.

[26] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON'99, 1999.

[27] Daniel Votipka, Seth M. Rabin, Kristopher Micinski, Thomas Gilray, Michelle L. Mazurek, and Jeffrey S. Foster. User comfort with Android background resource accesses in different contexts. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 235–250, Baltimore, MD, 2018.

[28] Yang Wang, Jun Zheng, Chen Sun, and Srinivas Mukkamala. Quantitative security risk assessment of Android permissions and applications. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 226–241. Springer, 2013.

[29] Takuya Watanabe, Mitsuaki Akiyama, Tetsuya Sakai, and Tatsuya Mori. Understanding the inconsistencies between text descriptions and the use of privacy-sensitive resources of mobile apps. In *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*, pages 241–255, 2015.

[30] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android permissions remystified: A field study on contextual integrity. In *USENIX Security Symposium*, pages 499–514, 2015.

[31] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. The feasibility of dynamically granted

permissions: Aligning mobile privacy with user preferences. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 1077–1093. IEEE, 2017.

[32] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Software engineering (ICSE), 2015 IEEE/ACM 37th IEEE international conference on*, volume 1, pages 303–313. IEEE, 2015.