Studying Permission Related Issues in Android Wearable Apps

Suhaib Mujahid, Rabe Abdalkareem and Emad Shihab
Data-Driven Analysis of Software (DAS) Lab
Concordia University, Montreal, Canada
{s_mujahi, rab_abdu, eshihab}@encs.concordia.ca

***Abstract*—Wearable devices are becoming increasingly popular; these devices host software that is known as wearable apps. Wearable apps could be packaged alongside handheld apps, hence they must be installed on the accompanying device (e.g., smartphone). This device dependency causes both apps to be also tightly coupled. Most importantly, when a wearable app is distributed by embedded it in a handheld app, Android Wear platform requires to include the wearable permission also in the handheld app which is error-prone.**

In this paper, we defined two permission issues related to wearable apps–namely permission mismatches and superfluous features. To study the permission related issues, we propose a technique to detect permission issues in wearable apps. We implement our technique in a tool called PERMLYZER, which automatically detects these permission issues from an app's APK. We run PERMLYZER on a dataset of 2,724 apps that have embedded wearable version and 339 standalone wearable app. Our result shows that I) 6% of wearable apps that request permissions are suffering from the permission mismatching problem; II) out of the apps that requires underlying features, 523 (52.4%) of handheld apps and 66 (80.5%) of standalone wearable apps have at least one superfluous feature; III) all the studied apps missed a declaration of underlying features for one or more of their permissions, which shows that developers may not know the mapping between the permissions they request and the hardware features. Additionally, in a survey of wearable app developers, all of the developers that responded mention that having a tool like PERMLYZER, that detect permission related issues would be useful to them. Our results contribute to the understanding of permissions related issues in wearable apps, in particular, proposing a technique to detect permission mismatch and superfluous features.

I. INTRODUCTION

Mobile apps are playing an increasingly important role in our daily lives. These mobile apps can monitor all kinds of activities. More recently, wearable devices that run wearable apps have enhanced the capabilities of these mobile apps. These wearable apps work closely with mobile apps and can significantly enhance the user experience.

The close interaction between mobile and wearable apps introduces its own unique challenges. One particular challenge is keeping the two apps (mobile and wearable) consistent in terms of the permissions they require. Permissions are used to control what an app can access, hence, mobile apps need to contain a superset of the permissions its associated wearable app requires. For example, if a wearable app needs access to the camera or microphone, it needs to explicitly ask for this permission in its connected mobile app configuration files (i.e., AndroidManifest.xml file).

Prior work by Au et al. [1], Stevens et al. [2], and Jha et al. [3] showed that the management of permissions is complicated and that permissions tend to be commonly misused by developers. Other work analyzed the permissions for apps published on Google Play store and focused on explaining the permissions usage and its implications on security and privacy [4, 5, 6, 7], permissions mis-usage [2, 1, 8], and suggesting which permission should be requested [9, 10, 11]. In fact, all of the aforementioned work focused on the permissions of mobile apps exclusively.

To make the situation even worse, the introduction of wearable devices and apps further complicates the permission management specially for devices running on Android version with API level lower than 23, which represent 50% of all the devices that visited the Google Play store [12]. The mobile (i.e., handheld) and wearable apps need to be in sync when requesting permissions (i.e., the wearable app permissions need to also be requested by its associated mobile app). This permission model is error-prone which may lead to installation and device compatibility issues. Thus, it impacts the success of wearable apps in the market [13]. However, to the best of our knowledge, no work has examined the permission issues related to wearables apps.

Therefore, in this paper, we conduct an empirical study to examine the permission related issues in the context of wearable apps. In particular, we focus on two of the most common issues. First, *permission mismatches*, which refer to the case where the permissions declared in the mobile and the wearable versions of the app do not match. Mismatched permissions can cause the wearable app to fail in the installation step or throw a security exception [13]. Second, *superfluous features*, which refer to the case where an app declares a permission that requires a specific hardware resource (e.g., access to the microphone) but the app does not use that permission. Having superfluous features can cause the Google Play store to filter out devices that do not support/have the hardware feature, reducing the potential customer base for the app [14].

We perform our study on 3,063 (2,724 embedded wearable apps and 339 standalone wearable apps) free apps from Google Play store that contain a wearable version. Our findings show that the permission mismatch issues exist in 6.1% of the analyzed released wearable apps that request permissions in

our dataset. Moreover, we find that 19.2% of the studied wearable apps contain superfluous features. To operationalize our work we built a tool, called PERMLYZER, that automatically detects these two permission related issues from Android APKs. PERMLYZER can be leveraged by developers to ensure that their app APKs do not suffer from any permission related issues prior to release. We also survey developers to better understand and evaluate the importance of our approach of detecting permission related issues in wearable apps. All responses to our survey indicate that having a tool like PERMLYZER that detects permission related issues would be useful to them.

This paper makes the following contributions:

- To the best of our knowledge, this is the first study to examine permission related issues in the context of wearable apps.
- We define and examine the two main permission related issues in the context of wearable apps–namely the permission mismatch problem and superfluous features. We then perform an empirical study to examine the prevalence of these permission related issues by investigating 3,063 wearable apps.
- We implement our approach in a tool called PERMLYZER, which is freely available. Also, to ease replication, all the data used in our study are publicly available [15].

The remainder of this paper is organized as follows: Section II provides a background about Android platform and wearable related concepts. Section III describes the issues that are related to wearable apps permission model. Section IV illustrates our study setup & approach. In Section V, we show the findings of our empirical study, and we discuss our findings in Section VI. We discuss the limitations of our study in Section VIII. Lastly we conclude our paper in Section IX.

II. BACKGROUND

Since wearable apps are fairly new, in this section we present a brief background on the development of wearable apps and the Android platform.

A. Android Platform and Distribution of Wearable Apps

Android is an open source platform that runs on different types of devices, including but not limited to, wearables, phones, tablets, televisions, and cars [13]. Android apps are distributed mainly through the Google Play store as APK archive files. Every APK must contains a configuration file in its main directory called `AndroidManifest.xml`. This file provides the necessary information about the app to the Android platform. Among other things, the manifest file does the following: 1) it has a description of the app components, 2) it contains the required list of the used permissions, 3) it has the declarations of the required hardware or software features, and 4) it specifies the minimum and target API level [16].

The Android platform also provides a framework of application programming interfaces (APIs) that apps use to interact with the underlying functionality of the platform (e.g., `CAMERA` and `MICROPHONE`). Each Android platform version is assigned a unique integer identifier, called the API level. Whenever a new platform version releases with an API change, the API level changes to a higher number. The new API remains compatible with all earlier API levels. Therefore, apps that are designed for a specific API level can run on a higher level, but it cannot run on a lower level [17].

Wearable apps on the Android platform can be distributed in two ways: 1) by embedding the wearable app inside a handheld app; or 2) by publishing more than one APK under the same app listing, using the multiple APKs feature of the Google Play store. When a user installs the handheld app, the Android platform will automatically install the wearable app on the paired wearable device [18, 19].

B. The Concept of Permissions in Android Platform

A permission is a restriction that limits access to sensitive data or dangerous device functionalities. The limitation is imposed to protect critical data and functionalities that could be misused to distort or damage the user experience [16]. Thus, developers request permissions to have access to sensitive data or high risk device functionalities. These permissions are declared in the `AndroidManifest.xml` file by adding the `<uses-permission>` element and specify the permission name in the attribute `android:name`. Line 2 in Listing 1 is an example of declaring a permission to read the received SMS.

Permissions have a protection level to characterize the potential risk implied in the permission. It indicates the procedure that the Android platform follows when determining whether or not to grant the permission to an app requesting it. The Android platform automatically grants permissions from the *Normal* protection level (e.g., `BLUETOOTH`) at installation, without asking for the user's explicit approval; and *Dangerous* permissions (e.g., `CAMERA` and `MICROPHONE`) that are requested by an app are displayed to the user and require confirmation before they are granted. Also, third-party apps can ask for permissions from both, the *Normal* and *Dangerous* protection level categories.

Listing 1: An example of the `AndroidManifest.xml` file.

```
1  <manifest . . . >
2  <uses-permission android:name="android.
       permission.READ_SMS" />
3  <uses-feature android:name="android.hardware.
       telephony"              android:required="
       true" />
4  . . .
5  <application . . .>
6  <activity android:name="com.example.project.
       FreneticActivity"     android:permission="
       com.example.project.DEBIT_ACCT" . . . >
7  . . .
8  </activity>
9  </application>
10 </manifest>
```

Specific to wearable apps, developers have to match permissions that are requested in the wearable version with permissions requested in the handheld version. In other words, all requested wearable permissions have to also be listed in the

manifest file of the handheld app [18]. However, the release of Android 6.0 (API level 23) introduced some major changes in the permission model; 1) apps that target and run on API level 23 or higher require users to grant their permission at the runtime instead of grant all the permission at once upon installation; and 2) wearable apps cannot receive permissions granted to the handheld apps [20, 21]. These changes affect how wearable app permissions are declared.

C. App Compatibility

The Android platform is designed to run on different types of devices, from wearables to phones and tablet devices. This range of devices provides a huge potential audience for an app. The Android devices have many different configurations, such as different hardware features, software features, the platform version, and screen configurations. To reach the largest possible user-base for an app, developers attempt to support as many device configurations as possible. Unfortunately, supporting all device configurations is impossible. In order to manage an app's availability based on device features, the Android platform defines feature IDs for hardware and software features that may not be available on all devices. Thus, developers can restrict their app's availability to devices through the Google Play store based on the device characteristics [22, 23]. When a developer uploads an app to the Google Play store, the store scans the app's manifest file and evaluates its elements such as the platform API level, declared features and requested permissions to establish the set of required features. On the user side, when a user browses an app on the Google Play store, the store compares the features that the app declared to the features available on the user's device to determine compatibility with the available devices [22]. Based on the previous process, the store decides whether the app is available to install on the user's device or not.

Developers declare all hardware and software features that their apps depends on in the `AndroidManifest.xml` file. The developers declare the features that their apps depends on by adding `<uses-feature>` element to the manifest file. This element has two main attributes: 1) `android:name`, to specify the name of the feature; and 2) `android:required`, to specify whether the app requires and cannot function without the declared feature, or whether it prefers to have the feature but can function without it [14]. Line 3 in Listing 1 is an example of a feature declaration for an app that depends on telephony functionalities.

For apps that request permissions that depend on hardware features, the Google Play store assumes that the apps use these underlying features and therefore requires the features even if there is no explicit mention of the required features in the manifest file. For such permissions, the Google Play store adds the underlying features to the metadata that it stores for every app and sets up filters for them. For example, if an app requests the `RECORD_AUDIO` permission but does not declare an `<uses-feature>` element for `android.hardware.microphone`, the Google Play store considers that the app requires a microphone and should not be shown to users whose devices do not have a microphone [14]. To avoid setting filters for hardware features that the app can operate without them, the app developer must declare the underlying features in the manifest file and give the value `false` to the attribute `android:required`.

III. Permission related issues

In this section, we illustrate the challenges of dealing with permission related issues in the context of the wearable app development. First, we enhance the discussion by presenting a motivating example. Second, we present the main two wearable apps permission related issues that are addressed in our study.

Motivating Example: According to the wearable permission model, wearable apps should match their handheld and wearable permissions. For example, if a wearable app needs to have the functionality to send a SMS, the app requests the permission `SEND_SMS`. As a response to the permission matching requirement, the handheld app should request the same permission even if it does not need it. When a user installs the handheld app, the Android platform installs the wearable app on the paired wearable device. Subsequently, the wearable app, which needs to support devices that run API level lower than 23, inherits the permissions that the platform granted to the handheld app. Failing to match the permissions can cause problems since the wearable app cannot get the required permissions. On the other hand, if the handheld app requests the permission `SEND_SMS`, the Google Play store will consider it as depending on a telephony functionality even if the app does not request the feature `android.hardware.telephony`. Hence, the handheld app will not be available in devices that do not have the telephony functionality, e.g., most of tablet devices.

In the following subsections, we discuss how wearable app permission related issues may affect wearable apps. We focus on the following two permission related issues: 1) Permission mismatches between the handheld and wearable apps; and 2) Missing the declaration of underlying features of the requested permissions.

A. Permission Mismatches

Description: To distribute a wearable app to users, the wearable APK can be embedded in a handheld APK. Then, when a user installs the handheld app, the Android platform pushes the embedded wearable app to the paired wearable device. If the user grants the permissions to the handheld app during the installation process, the wearable app inherits the granted permissions from the handheld app. To ensure that the user grants the wearable app's permissions, developers are required to match the wearable app's permissions with the handheld app's permissions by including all the permissions declared in the wearable's manifest into the handheld's manifest. This process should be performed even if the handheld app does not use those permissions [24]. In case the permissions are not requested properly, i.e., a wearable version of an app may request a permission that is not requested by the handheld

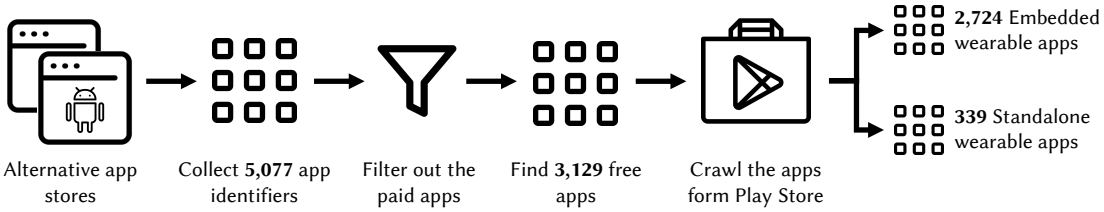


Fig. 1: Overview of the data collection process.

version of the app. We call this case ***permission mismatch*** problem.

Implication: As a result, a wearable app that suffers from the permission mismatch problem cannot grant its permissions which may lead to one of the following problems; 1) the wearable app fails to be installed on the wearable device, 2) throws a security exception and/or crash the app [25]. Additionally, the permission mismatch problem is particularly problematic since: 1) it does not raise compilation errors or print any log messages; 2) it runs normally on the emulator or any wearable devices using Android Debug Bridge (adb); 3) it is not automatically detected as a problem by the IDEs, including Android Studio; and 4) it is hard to catch since it affects only the devices that run with API level lower than 23. Hence, the permission mismatch problem is usually caught by users.

B. Superfluous Features

Description: Feature declarations in the manifest file are informational only, which means that the Android platform does not check them before installing an app. However, some app stores such as, Google Play, checks the feature declarations when it interacts with apps. According to the Android developer documentation, missing a feature declaration used by an app should be considered an error [14].

Features and permissions that an app declares in the manifest file can affect the filtering step that the Google Play store performs. The Google Play store uses features and permissions to determine whether an app is compatible with a device, or the app depends on features that are not available on the device. The app store checks the permissions in the manifest file of each app and sets up filters for apps that have permissions which require underlying features even if they are not declared. Thus, requesting a permission in the manifest file causes the Google Play store to set filters for features that the app does not depend on. As a result, the app store filters out the app from compatible devices. Hence, for permissions that depend on underlying features, developers must explicitly specify in the manifest file whether the app cannot function without the underlying feature, or whether it prefers to have the feature but can function without it.

Implication: In case an app misses to declare an underlying feature, the Google Play store automatically adds the feature to the metadata that it stores for the app. Based on the metadata, the store sets up filters for the features even if the app can handle the absence of such features. Moreover, the underlying features could belong to unused permissions, which the app requests without using the functionalities that they grant to the app - we call such a case the ***superfluous feature***. As a result, the superfluous features unexpectedly filters out legitimate compatible devices, which reduces the potential customer base for an app, and negatively impacts its revenues.

IV. Study Setup & Approach

In this section, we detail our dataset collection, and present PERMLYZER, an automated approach to detect permission related issues. Figure 2 describes our automated approach and the following subsections detail our approach.

A. Dataset

As shown in Figure 1, we select the available Android Wear apps in the Google Play store by collecting their identifiers from two alternative app markets: *Android Wear Center* [26] and *GoKo Store* [27], accessed on July 7th, 2017. By filtering out paid apps from the set of 5,077 apps, we were able to collect 3,129 free apps. We focus on free apps since we need to download and unpack the apps. In order to download the last version of the selected apps, we built a crawler that interfaces with the Google Play store's API as a regular mobile device to download the handheld apps and as a wearable device to download the standalone wearable apps. The apps' APKs were collected from July 19th through 21st, 2017. We were able to download and unpack 3,063 apps, since some of the apps are not available on Google Play store anymore. After downloading and unpacking the apps, we find 2,724 apps contains an embedded wearable app and 339 apps have a standalone wearable app.

B. Detecting Permission Mismatches

To detect permission mismatches, we start by extracting the embedded wearable app from the handheld app's APK. Then, we extract the permissions from both embedded and handheld APKs. Next, we identify if the permission model of the wearable app requires matching the permissions; if so, all permissions in the embedded wearable app should be requested in the handheld app. Finally, we detect the permission mismatches by examining the permission of the wearable and the handheld app. The following subsections are the detailed steps to automatically analyze APK files of an wearable app and detect the permission mismatch problem. Figure 2 (Section ①) illustrates the automated approach overview.

1) Extract the Embedded Wearable APK: In order to extract the APK file of the embedded wearable app, we first unpack the handheld app's APK and decode the resources using APKTOOL [28], a tool for reverse engineering Android apps.

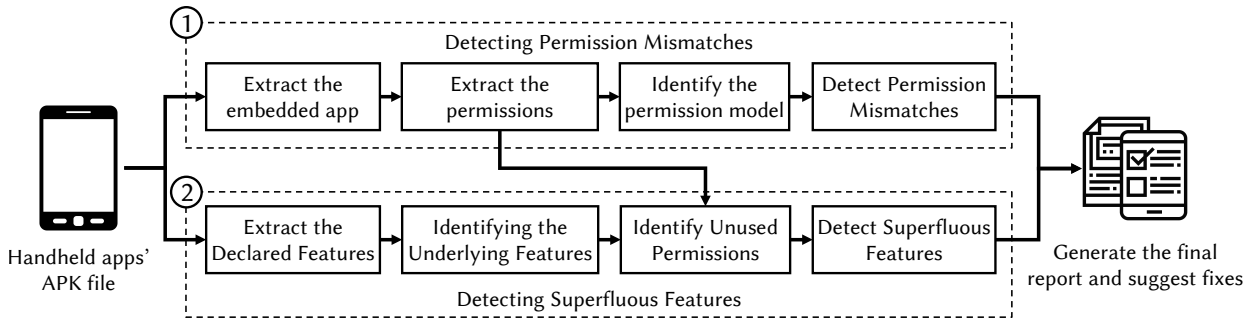


Fig. 2: Approach overview.

The tool allows us to retrieve the original form of the XML files. After we obtain the unpacked resources for the handheld app, we need to identify the path to the wearable version's APK file, so we apply the following steps: 1) extract and parse the XML tree of `AndroidManifest.xml` file from the main directory, 2) select the metadata tag that refers to the wearable app description file[1] by targeting the tag name `com.google.android.wearable. beta.app`, and 3) parse the XML tree for the description file and extract the path of the wearable APK by targeting the `rawPathResId` tag. In some cases, a handheld manifest file has a configuration mistake, e.g., missing the path of the wearable app description file, or incorrect APK path could cause a failure in detecting the wearable APK. In such case, we use the `MANIFEST.MF` file to detect the path of the wearable APK.

Every Java package has the file `MANIFEST.MF` as a default manifest, which is stored in the `META-INF` directory, the default manifest used to define extensions and package-related data, such as the list of files and their paths. Since the APK file of the wearable app is packaged inside its handheld app APK, we use regular expressions to search for paths of all files with `.apk` extension from the `MANIFEST.MF` file. In the case of multiple APK files, we extract and unpack them to figure out which ones belong to the wearable app. We distinguish the wear app's APK based on three heuristics, which include: 1) matching the package ID of the embedded and handheld apps, 2) looking at the name of the APK file that contains keywords such as *'wear'*, and/or 3) looking for the usage of tags that indicate the use of wearable hardware in the manifest file, e.g., `android.hardware.type.watch`.

2) Extract the Requested Permissions: First, we parse the file `AndroidManifest.xml` for both of the handheld app and the embedded wearable app. Second, for both manifests, we select the permissions by identifying the tags `<uses-permission>`; then for each tag we read the value of the attribute `android:name`. Finally, we check if the permission(s) belongs to the Android Open Source Project (AOSP) or to a third party app. We distinguish between permissions related to AOSP and third party apps since: 1) The Android platform does not check for the matching of permissions of third-party apps, 2) we cannot check if these permissions are used or not because we do not know the corresponding APIs for these permissions.

3) Identifying the Permission Model: To detect the permission mismatches, we automated performing the following steps. First, we check which permission model the app should implement. To do so, we select the tag `<uses-sdk>` from the manifest file of the handheld app and we read the value of the attribute `android:minSdkVersion`. If the value is lower than 23, then the app should match the permissions of the embedded wearable app version with the handheld version's permissions.

4) Detecting Permission Mismatches: Based on the previous step, if the app is required to match the permissions, we perform the process of detecting the permission mismatches. The process starts by matching the two permissions lists that we extracted from the handheld and the embedded apps. For each permission requested in the embedded wearable app, we check if it exists in the list of requested permissions in the handheld app; if not, we report it as a permission mismatch.

C. Detecting Superfluous Features

To detect superfluous features, we start by extracting the declared features form the manifest file. Next, we identify the underlying features for the requested permissions. Then, for each underlying feature, we check if the app is actually using the corresponding permissions. Finally, we report the underlying feature that belong to unused permissions as superfluous features. Figure 2 (Section ②) illustrates the overview of our automated approach.

1) Extract the Declared Features: To extract the features that the app declared in its manifest file, we use a similar approach to the process of extracting the requested permissions that we illustrated in Section IV-B2. We target the tags `<uses-feature>` in the manifest file; then for each tag, we extract the value of two attributes: 1) the attribute `android:name` to get the name of the feature, and 2) the attribute `android:required` to check if the app is designed to function without the feature or not.

2) Identifying the Underlying Features: In order to detect the missed underlying features for an app, we start by identifying the underlying feature for all requested permissions in the app's manifest file. We depend in this step on a tool

[1]A file that contains the version and path information of the wearable app APK.

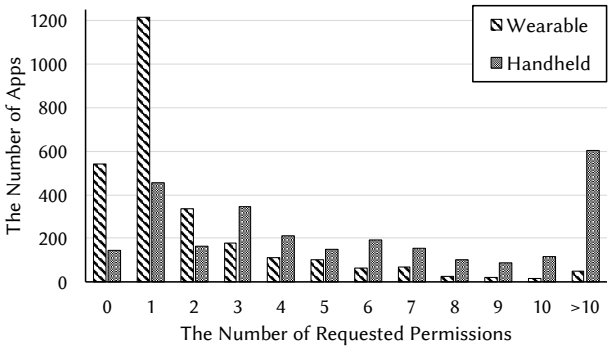


Fig. 3: Histogram of the number of wearable and handheld apps at each level of requested permissions.

called APKANALYZER [29] from the Android's Software Development Kit (SDK). Among other things, this tool extracts features that trigger the filtering of the Google Play store. The output of this tool contains both of the features that are already declared in the manifest file and the underlying features of requested permissions. Also, if applicable, the tool's output includes a mapping between the underlying features and the corresponding permissions. Finally, if a feature in the output of the APKANALYZER does not exist in the list of declared features that we extracted in the previous step, we report it as a missed underlying feature.

3) Identify Unused Permissions: In order to use some APIs, the Android Platform requires permissions. For example, calling an API to access the microphone requires an internal check by the Android platform for the permission to ensure that the app has the permission `RECORD_AUDIO`.

To check which permissions an app uses, we need: 1) a mapping between every public Android platform API and the required permissions to use that API; and 2) the list of all platform's API calls that the app performs. Then, we link the list of API calls that the app performs with the required permissions for these APIs.

The permission mapping in our approach is based on PSCOUT [1], a tool that extracts the permission specifications from the Android platform source code using a static source code analysis. On the API calls side, we depend on the tool Androguard [30] to extract all possible API calls from apps.

At the end of this step, we have two lists of permissions: 1) the list of AOSP's permissions that the app requests, which we extracted them in Section IV-B2 ; and 2) the used permissions by linking the API calls to their required permissions. The requested permissions that does not appears in the list of used permissions are considered as ***unused permissions***.

4) Detecting Superfluous Features: Depending on the previous steps, for each missed underlying feature, we checked its corresponding permission. If the permission does exist in the list of unused permissions, we report its corresponding feature(s) as a superfluous feature.

D. Generate the Final Report & Suggest Fixes

To operationalize our work we developed a tool, called PERMLYZER (**Perm**issions Ana**lyzer**), that automatically detects the permission mismatches and the superfluous features. The tool takes as input, a handheld app's APK file of wearable app or an APK file of standalone wearable app. Based on the detected problems the tool generates suggestions to address the problems. Also, the tool auto-generates a new `AndroidManifest.xml` file that implements the suggested fixes. Although the tool is able to automatically generate a corrected AndroidManifest.xml, we are not able to regenerate the app's APK since each individual APK needs a specific certificate, which we do not have.

TABLE I: THE MOST MISMATCHED PERMISSIONS IN THE STUDIED APPS.

Permission Name	Mismatch (%)
READ_CALENDAR	12.10
WAKE_LOCK	11.40
ACCESS_FINE_LOCATION	8.30
RECEIVE_COMPLICATION_DATA	7.60
VIBRATE	7.60
READ_PHONE_STATE	6.80
WRITE_EXTERNAL_STORAGE	6.80
READ_EXTERNAL_STORAGE	6.10
ACCESS_NETWORK_STATE	3.80
BODY_SENSORS	3.80
BLUETOOTH	3.00
INTERNET	2.30

V. RESULTS

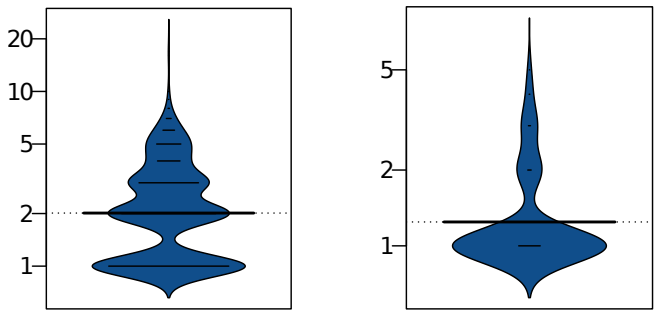
The main *goal* of our study is to detect and examine permission related issues in the context of wearable apps. Although some very recent work examined the permission problems in mobile apps [31], to the best of our knowledge, this is one of the first studies to exclusively focus on the permission related issues in the context of wearable apps. In addition to defining the main issues related to permission in wearable apps, we also examine and quantify these permission issues in a dataset of apps published on the Google Play store. We formalize our study with the following two questions:

- *RQ1: How widely does the permission mismatch problem exist in wearable apps?* In this question we want to check the existence of the permission mismatch problem in the published apps, so we run PERMLYZER on the 2,724 wearable apps. The tool checks if the app requires matching the handheld/wearable permission, if so, the tool reports the permission mismatches.
- *RQ2: How do wearable apps declare the hardware features of their permissions?* To better understand the problem of missing to declare the underlying features in real word apps, we use PERMLYZER to analyze the handheld and stand alone wearable apps and detect the missed underlying features.

For each research question, we provide a motivation, approach, and discuss the result.

RQ1: How widely does the permission mismatch problem exist in wearable apps?

Motivation: Since it is up to the developers to match the wearable app permissions with the handheld app permissions,



(a) Handheld (b) Standalone wearable

Fig. 4: The number of missed underlying features in the studied apps.

developers may mismatch these permissions. Hence, our goal is to quantify the number of wearable apps that suffer from these permission mismatch problem.

Approach: To answer this research question, we use the tool PERMLYZER that internally applies the approach that we described in Section IV-B to detect the permission mismatches. Matching the permissions is required when the wearable app requests permissions that are not requested in it corresponding handheld app. So, we discard the apps that do not request any permissions in their wearable version from our analysis. Figure 3 shows the number of requested permissions in the studied wearable apps. Out of all embedded wearable apps, 541 apps do not request any permissions at all. Thus, they are not required to match their permissions with the wearable permissions. This filtering left us with a set of 2,178 apps which we analyze to detect permission mismatches. We also investigate what type of permissions are most likely to be mismatched.

Findings: The results show that all the 2,178 wearable apps in the analyzed dataset are built to be compatible with platform versions that require matching the permission between wearables and handhelds (i.e., Android API versions below 23). We find that 132 (6.1%) of the examined apps suffer from the permission mismatch problem. Of these 132 apps, the number of mismatched permissions ranges between 1 to 6 permission mismatches, with a median of one mismatched permission per app. An example, `AutomateIt.mainPackage` app requests the permission `READ_EXTERNAL_STORAGE` in the wearable app, but does not request the same permission in the handheld version.

Table I shows the mismatched permission types and the number of cases for each of them. We observe that the most commonly missed permissions are related to access the calendar, power manager, and location.

Out of the 2,178 apps that request permissions in its embedded wearable apps, 6.1% suffer from the permission mismatch problem in their last released version on Google Play store.

TABLE II: THE UNDERLYING FEATURES WITH THE PERCENTAGE OF HANDHELD AND STANDALONE WEARABLE APPS THAT DECLARED/MISSED THE UNDERLYING FEATURES.

Feature Name	Handheld (%)		Standalone (%)	
	Missed	Declared	Missed	Declared
bluetooth	27.5	-	4.8	-
camera	3.0	-	-	-
location	62.9	0.2	2.5	-
location.gps	4.0	-	-	-
location.network	3.1	-	-	-
microphone	12.4	0.1	0.8	-
telephony	8.5	-	-	-
wifi	42.4	0.2	1.2	-

RQ2: How do wearable apps declare the hardware features of their permissions?

Motivation: In this research question, first we want to study how apps declare their hardware features for their requested permissions. Thus, we examine the use of this functionality in the published wearable apps on the Google Play store. Second, we study the missed underlying features and examine if they are superfluous features or not.

Approach: We run the PERMLYZER tool on the 2,724 handheld apps and the 339 standalone wearable apps. Since, we focus on the declaration of hardware features, we exclude apps that do not request permissions that depend on a hardware feature. So, we end up with 999 handheld app and 82 standalone wearable apps.

Findings: We find that all the studied handheld and standalone wearable apps missed a declaration of underlying features for one or more of their permissions. For example, the handheld app `slide.watchFrenzy` requests the permission `ACCESS_WIFI_STATE` without declaring its underlying feature `android.hardware.wifi`. Moreover, while 28.3% of the apps declare at least one hardware feature, we find that only 5 apps out of the 999 handheld apps declare any underlying features for their permissions; and none of the standalone wearable apps declare any underlying features. This shows that developers may not know the mapping between the permissions they request and the hardware features. Figure 4 shows the count of missed underlying features in both of the handheld apps and embedded wearable apps. On median the studied handheld apps missed to declare 2 hardware features and 17 at max. For the standalone wearable apps, on median they missed 1 hardware feature and at max 5 features.

To emphasize the underlying feature declarations, Table II shows the features (Column 1) with the percentage of apps that declared it as underlying feature (Column 3 & 5) and the percentage of apps that missed to declare the feature (Column 2 & 4). We observe that the handheld apps mostly missed the declaration of location, wifi, and bluetooth feature. For the standalone wearable apps, bluetooth, camera, wifi, and microphone are the missed underlying features. For example, the standalone wearable app *com.jeremysteckling.facerrel* requests `ACCESS_WIFI_STATE` permission without declaring its underlying feature `android.hardware.wifi`. As a

TABLE III: LIST OF UNUSED PERMISSIONS THAT INTRODUCED SUPERFLUOUS FEATURES WITH THE PERCENTAGE OF AFFECTED HANDHELD APPS.

Permission Name	Feature Name	Apps (%)
ACCESS_WIFI_STATE	wifi	14.3
BLUETOOTH	bluetooth	12.7
BLUETOOTH_ADMIN	bluetooth	10.7
ACCESS_COARSE_LOCATION	location	9.4
ACCESS_FINE_LOCATION	location	6.4
CHANGE_WIFI_STATE	wifi	6.4
READ_SMS	telephony	5.2
RECORD_AUDIO	microphone	4.4
RECEIVE_SMS	telephony	3.2
CHANGE_WIFI_MULTICAST_STATE	wifi	1.9
CAMERA	camera	1.4
PROCESS_OUTGOING_CALLS	telephony	1.3
ACCESS_LOCATION_EXTRA_COMMANDS	location	1.1
WRITE_SMS	telephony	0.8
SEND_SMS	telephony	0.7
ACCESS_FINE_LOCATION	location.gps	0.5
ACCESS_MOCK_LOCATION	location	0.5
ACCESS_COARSE_LOCATION	location.network	0.4
RECEIVE_MMS	telephony	0.4
RECEIVE_WAP_PUSH	telephony	0.2
WRITE_APN_SETTINGS	telephony	0.2
CALL_PRIVILEGED	telephony	0.1
MODIFY_PHONE_STATE	telephony	0.1

result, the app is not available for the wearable devices that do not support the Wi-Fi connectivity.

Also, we found that 523 (52.4%) of the handheld apps and 66 (80.5%) of the standalone wearable apps have at least one superfluous feature. An example is the app com.runar. wearcompass, which requests the permissions ACCESS_COARSE _LOCATION and ACCESS_FINE_LOCATION without using their corresponding APIs. At the same time, the app does not declare that it does not depend on the feature android.hardware.location which make Google Play store filters the app from devices that do not provide the functionality of detecting the location.

Table III shows the list of permissions that introduce the superfluous feature in the studied handheld apps; for each one of them, we calculate the percentage of affected apps. From this table we can see that the highest percentage of apps are affected by superfluous features that were introduced by the permissions ACCESS_WIFI_STATE, BLUETOOTH, and BLUETOOTH_ADMIN with percentage value of 14.3%, 12.7%, and 10.7% respectively. For standalone apps, Table IV shows that the most superfluous features are introduced by the permission BLUETOOTH with 50.0% and the permission ACCESS_FINE_LOCATION with 18.3%. We observe that the highest percentage of apps in both of the handheld and standalone wearable apps are affected by superfluous features caused by permissions related to network communication and location detection.

Out of the apps that requires underlying features, 523 (52.4%) of handheld apps and 66 (80.5%) of standalone wearable apps have at least one superfluous feature.

TABLE IV: LIST OF UNUSED PERMISSIONS THAT INTRODUCED SUPERFLUOUS FEATURES WITH THE PERCENTAGE OF AFFECTED STANDALONE WEARABLE APPS.

Permission Name	Feature Name	Apps (%)
BLUETOOTH	bluetooth	50.0
ACCESS_FINE_LOCATION	location	18.3
ACCESS_WIFI_STATE	wifi	6.1
BLUETOOTH_ADMIN	bluetooth	6.1
ACCESS_COARSE_LOCATION	location	2.4
CHANGE_WIFI_STATE	wifi	2.4
CHANGE_WIFI_MULTICAST_STATE	wifi	1.2
RECORD_AUDIO	microphone	1.2

VI. DISCUSSION

In this section, we examine the context of our findings. First, we discuss the problem of unused permissions in the wearable apps. Then, we discuss the relation between the wearable permission model and detection overprivileged permission in handheld apps. Finally, we discuss the developers' perspective about permission related issues in wearable apps.

A. The Relation between Permissions Mismatch and Unused Permissions.

Throughout the evolution of an app, developers may introduce and remove different permissions. Previous work showed that mobile apps tend to have more overprivileged permissions (i.e., apps that ask for more permissions than they require/need) [32, 33]. As we describe in the Section II-B, permissions requested in an embedded wearable app may need to be requested in the corresponding handheld app as well. Since wearable and handheld apps are in two separated modules; developers have to reflect changes in multiple places. This permission model could be error-prone and increase the chance to leave more unused permissions in the manifest file.

To understand how the requirement of matching the wearable permissions affects the amount of unused permissions in handheld apps, we examine the amount of unused permissions in the handheld apps that are requested in their embedded wearable apps. We used PERMLYZER to extract all unused permissions from the handheld apps. Next, for each app the tool extracts the embedded wearable app and checks if the unused permissions are requested in the embedded version.

The results showed that; 1) 56.2% (1,532) of handheld apps have unused permissions; 2) by analyzing their embedded wearable apps, we observe that 24.2% (371) of handheld apps with unused permissions are requesting all the unused permissions in their wearable version. Furthermore, we find 44.3% (678) of them are requesting at least one of their unused permissions in the wearable version. For example, the app com.ppltalkin.findmywatch requests the permissions WAKE_LOCK and RECORD_AUDIO in the handheld without using their corresponding APIs, however, the app request these permissions in its wearable version.

Felt et al. [8] studied 795 mobile apps and showed that 32.7% of them have unused permissions. More recently, Wei et al. [33] studied 1,703 app versions and found that 33.2% of them have unused permissions. By comparing the unused

permissions in the wearable apps, we find that wearable apps have about $1.7X$ more unused permissions. As the comparison shows, the permission matching requirement can be a factor to introduce unused permissions in handheld apps.

B. Overprivileged Permissions Vs. Unused Permissions.

Overprivileged permissions are permissions that apps request but their corresponding APIs never use. So, removing those permissions should not affect the app functionality. These overprivileged permissions could introduce vulnerabilities and raise security concerns [33].

Several studies analyzed the permissions for apps published on the Google Play store and focused on explaining the permission usage and its implications on security and privacy [4, 5, 6, 7], evolution over time [33, 32], discover permission misuses and overprivilegesd [2, 1, 8] and suggest which permissions should be requested [9, 10, 11]. To the best of our knowledge, previous studies do not consider the notion of wearable apps when they perform their analysis that deals with the problem of overprivileged permissions. So, permissions that are not mapped to an API call are considered as overprivileged; although for wearables, the handheld apps may hold unused permissions to satisfy the permission matching requirement. Hence, *not all of the unused permission necessary are overprivileged permissions*.

Our analyses shows that out of all apps that request unused permissions, only 55.7% of them have all of their unused permissions are legitimately privileged. And 24.4% of all unused permissions in wearable apps are legitimately privileged. The evidence shows that a high percentage of unused permissions in wearable apps are legitimately privileged. Thus, it is useful for follow up research to consider the notion of wearable apps in order to improve the accuracy of their results when they analyze such apps.

C. Wearable App Developers' Perspective.

Thus far, our analysis has been quantitative in nature. To triangulate our findings and better understand and evaluate the importance of our approach of detecting permission related issues in wearable apps, we perform a complementary qualitative analysis to understand: 1) the developers perspective about permission related issues in wearable apps and 2) if they consider our proposed technique to be useful.

To do so, we designed an online survey. The survey included four main sections: 1) three questions regarding the participant's background and experience in the development of wearable apps, 2) a Likert-scale question about the difficulty of catching permission related issues in wearable apps, 3) three questions asking if the developers knows that there are permission issues that exist in their wearable apps and how they discover them, and 4) if the developers thinks that having a tool like PERMLYZER (e.g., a plugin in your IDE) to identify permission related issues in wearable apps based on our proposed techniques would be useful.

To identify the target population, for apps that we determined to have permission related issues, we collected the developers' names and their email addresses and the version and name of their wearable apps. In total, we found 110 unique developers for 160 wearable apps. We then randomly selected 100 unique wearable apps developers and we successfully sent the survey to 82 of them (for some the email address bounced). We received 7 responses, i.e., the response rate is approximately 9%. Although this is a low number of responses, it is in line with, and even higher, than the typical 5% response rates reported in software engineering surveys [34].

Of the seven respondents, five identified themselves as having wearable app development experience between two to five years, and two have more than five years of wearable apps development experiences. All the participants claim to have more than two published wearable apps.

The majority of the developers (6 out of the 7) mentioned that catching permission related issues in wearable apps to be difficult. Only one participant indicated that catching permission issues to be a 'trivial task'. However, when we asked the participants whether they know that the identified permission issues existed in their apps, five participants stated that they know that their apps suffer from permission issues. While two participants do not know that their wearable apps have permission issues. For the participants who know about the permission issues in their apps only one developer fixed the permission issues and said that *"I was going to add a feature which had that permission but did not add the feature"*. One possible reason for that is the lack of an automated tool to detect and fix these permission issues. Finally, *all seven* of our survey participants mentioned that having a tool like PERMLYZER or plugin such as the one we propose will definitely be useful.

VII. RELATED WORK

In this section, we describe work that is related to our study. To the best of our knowledge no previous work exists that studies the permission related issues in wearable apps.

Recently, Mujahid et al. [35] studied the user complaints of wearable apps by analyzing 589 reviews from 6 Android wearable apps. One of their findings indicates that amongst other, users complain mostly about functional errors and missing notifications on wearable apps. Zhang and Rountev [36] presented formal semantics to statically model the notification mechanism of Android Wear and contributed with the development of two domain-specific tools, one for test case execution, and another for automated test generation. Ahola [37] exposed three issues and limitations found in the Android Wear platform during wearable app development that are: better wear Internet connectivity, virtual button support for watch faces, and software configurable language support for voice input. From a different perspective, Lyons [38] did a study on the user perceptions about the functionality and design of smartwatches, including android wearable devices. Chauhan et al. [39] did a previous categorization of smartwatch apps from Samsung, Apple and Android Wear. They used *Android Wear Center* [26] and *GoKo* store [27] as sources to get the wear app identifiers for crawling their information; we applied the same approach to initialize our crawling phase.

Other prior studies focused on permission issues in the context of Android apps [10, 9, 2]. For example, several studies found that issues and misuses of declaring app permissions are common in handheld apps [2, 40, 1]. Also, a number of studies proposed techniques that provide API to perform permission mappings in order to mitigate missed permissions [1, 8]. Jha et al. [3] studied mistakes in Android manifests for mobile apps, and found that more than 78% of the studied apps have at least one configuration error. Android `lint` [41] is an analysis tool built by Google that statically analyzes source code files, including manifest files. ManifestInspector [3] is one of the performing tools when it comes to detecting errors in Android manifest files; this tool's functionality is based on a number of effective rules. Currently, lint (in Android Studio 1.5) defines only 30 rules related to manifest files while ManifestInspector defines 116 rules. However, none of these rules target the specialty of manifest files' configuration for wearable apps. Wei et al. [33] conducted a study on the evolution of Android permissions, focusing on the differences between pre-installed and third party apps. They analyzed patterns and permission distributions, and reported that apps tend to be overprivileged and to request more permissions over time.

Our study differs from prior work since we focus on the inconsistent configuration problems that may exist between wearable and handheld versions of an app. More specifically, we study the permission related issues.

VIII. Threats to Validity

In this section, we disuses threats to validity of our study.

Threats to Internal Validity: Our results depend on the accuracy of the used tools. To detect the unused permission, the presented approach relies on ANDROGUARD [30] tool to extract the platform API calls using a static analysis approach and on the mapping of PSCOUT [1] tool to link the platform APIs with the corresponding permission. To help alleviate this threat, we manually investigated some of the reported results as unused permissions and in all cases the manually examined cases were correct. We also use APKTOOL [28] to retrieve the original `AndroidManifest.xml`. Our process is only as accurate as the APKTOOL. For instance, the retrieved XML file could be missing some comments, having extra elements that were added in the compilation phase and/or have different space formatting. That said, we examine some of retrieved XML files and found that they do not have any missing elements that could affect our approach. Our results also include only Android Open Source Project (AOSP) permissions in the process of analyzing unused permissions. Other third party permission could have different pattern in term of declare the unused permissions.

To extract the embedded wearable APKs, we rely on extracting the path of the wearable APK from the handheld Manifest file. However, in some cases, the handheld Manifest file may not have the path of the wearable APK file, or have an incorrect APK path. To deal with these cases, we developed three heuristics. Thus, our approach may produce false positive cases. To help alleviate this issue, we manually investigated some of the extracted cases and found that, in all cases, our heuristics extract the correct embedded wearable APK path.

To understand and evaluate the importance of our approach of detecting permission related issues in wearable apps, we conducted an online survey. We contacted 82 developers of wearable apps that we determined to have permission related issues and received 7 ($\sim$9%) responses. While this response rate may be considered small, it is acceptable in questionnaire-based software engineering surveys [34].

Threats to External Validity: We apply our techniques on free wear apps only, because of this, our results may not be generalizable to paid wearable apps. Also, our empirical study is based on apps that are already published on Google Play store. This means our results may not reflect the permission related issues in other app stores. The permission matching model are required only for wearable apps that need to support devices run API level lower than 23. However, a snapshot of data represents all the devices that visited the Google Play store shows that more than 50% of devices running on Android version with API level lower than 23 [12], which highlight the importance of supporting such devices.

IX. Conclusion

Wearable apps' popularity is increasing. In fact, based on our data collection, the Google Play store contains more than 5,077 wearable apps. In this paper, we defined two permission issues related to wearable apps–namely permission mismatches and superfluous features. To study the permission related issues, we propose a technique to detect permission issues in wearable apps. We implement our technique in a tool called PERMLYZER, which automatically detects these permission problems from an app's APK. We run PERMLYZER on a dataset of 2,724 apps that have embedded wearable version and 339 standalone wearable app. Our result shows that I) 6% of wearable apps that request permissions are suffering from the permission mismatching problem; II) out of the apps that requires underlying features, 523 (52.4%) of handheld apps and 66 (80.5%) of standalone wearable apps have at least one superfluous feature; III) all the studied apps missed a declaration of underlying features for one or more of their permissions, which shows that developers may not know the mapping between the permissions they request and the hardware features. In a survey of wearable app developers, all of the developers that responded mention that having a tool like PERMLYZER, that detects permission related issues would be useful to them.

The result in this paper outline some directions for future work. First, to gain practical feedback from developers about the advantages of using our tool (PERMLYZER), we plan to submit the fixed AndroidManifest.xml to the wearable app repository that suffer from these permission problems. Also, we plan to perform an in-depth investigation to understand what are the root causes that introduce these permission problems in wearable apps. Finally, understanding the relation between the domain of the wearable apps and the most common missed permissions is another interesting point for future work.

References

[1] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the ACM Conference on Computer and Communications Security*, ser. CCS '12. ACM, 2012, pp. 217–228.

[2] R. Stevens, J. Ganz, V. Filkov, P. Devanbu, and H. Chen, "Asking for (and about) permissions used by android apps," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, May 2013, pp. 31–40.

[3] A. K. Jha, S. Lee, and W. J. Lee, "Developer mistakes in writing android manifests: An empirical study of configuration errors," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 25–36.

[4] M. L. Dering and P. McDaniel, "Android market reconstruction and analysis," in *Proceedings of the IEEE Military Communications Conference*, ser. MILCOM '14. IEEE Computer Society, 2014, pp. 300–305.

[5] T. Watanabe, M. Akiyama, T. Sakai, and T. Mori, "Understanding the inconsistencies between text descriptions and the use of privacy-sensitive resources of mobile apps," in *Proceedings of Eleventh Symposium On Usable Privacy and Security*, ser. SOUPS '15, USENIX Association. USENIX, 2015, pp. 241–255.

[6] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal analysis of android ad library permissions," *arXiv preprint arXiv:1303.0857*, 2013.

[7] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. ACM, 2009, pp. 235–245.

[8] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. ACM, 2011, pp. 627–638.

[9] M. Y. Karim, H. Kagdi, and M. D. Penta, "Mining android apps to recommend permissions," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER '16, vol. 1. IEEE Press, March 2016, pp. 427–437.

[10] L. Bao, D. Lo, X. Xia, and S. Li, "What permissions should this android app request?" in *Proceedings of International Conference on Software Analysis, Testing and Evolution*, ser. SATE '16, Nov 2016, pp. 36–41.

[11] ——, "Automated android application permission recommendation," *Science China Information Sciences*, vol. 60, no. 9, p. 092110, Jul 2017.

[12] Android documentation, "Dashboards," https://developer.android.com/about/dashboards/index.html, 2017, accessed on October 31, 2017.

[13] S. Mujahid, G. Sierra, R. Abdalkareem, E. Shihab, and W. Shang, "An empirical study of android wear user complaints," *Empirical Software Engineering*, Mar 2018.

[14] Android documentation, "uses-feature," https://developer.android.com/guide/topics/manifest/uses-feature-element.html, 2017, accessed on October 6, 2017.

[15] S. Mujahid, R. Abdalkareem, and E. Shihab, "The dataset of wearable permission analyses," 2018. [Online]. Available: http://das.encs.concordia.ca/publications/wearable-permissions-data

[16] Android documentation, "App manifest," https://developer.android.com/guide/topics/manifest/manifest-intro.html, 2017, accessed on November 30, 2017.

[17] ——, "uses-sdk element," https://developer.android.com/guide/topics/manifest/uses-sdk-element.html, 2017, accessed on October 4, 2017.

[18] ——, "Packaging wearable apps," https://developer.android.com/training/wearables/apps/packaging.html, 2017, accessed on January 19, 2017.

[19] ——, "Standalone apps," https://developer.android.com/training/wearables/apps/standalone-apps.html, 2017, accessed on November 30, 2017.

[20] ——, "Requesting permissions at run time," https://developer.android.com/training/permissions/requesting.html, 2017, accessed on October 4, 2017.

[21] ——, "Requesting permissions on android wear," https://developer.android.com/training/articles/wear-permissions.html, 2017, accessed on October 31, 2017.

[22] ——, "Device compatibility," https://developer.android.com/guide/practices/compatibility.html, 2017, accessed on October 4, 2017.

[23] ——, "Multiple apk support," https://developer.android.com/google/play/publishing/multiple-apks.html, 2017, accessed on October 4, 2017.

[24] ——, "Packaging wearable apps," https://developer.android.com/training/wearables/apps/packaging.html, 2017, accessed on January 19, 2017.

[25] S. Mujahid, "Detecting wearable app permission mismatches: A case study on android wear," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '17. ACM, 2017, pp. 1065–1067.

[26] Wearable Software, "Android wear center," http://www.androidwearcenter.com, 2016.

[27] J. Korner, L. Hitzges, and D. Gehrke, "Goko," http://goko.me, 2016.

[28] C. Tumbleson and R. Winiewski, "Apktool - a tool for reverse engineering 3rd party, closed, binary android apps." https://ibotpeaches.github.io/Apktool/, 2017, accessed on May 4, 2017.

[29] Android Studio, "Apk analyzer tool," https://developer.android.com/studio/command-line/apkanalyzer.html, 2017, accessed on December 1, 2017.

[30] A. Desnos and G. Gueguen, "Androguard: Reverse engineering, malware and goodware analysis of android applications)," https://github.com/androguard/

androguard, 2017, accessed on November 27, 2017.

[31] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 817–847, Sept 2017.

[32] P. Calciati and A. Gorla, "How do apps evolve in their permission requests? a preliminary study," in *Proceedings of 14th IEEE/ACM International Conference on Mining Software Repositories*, ser. MSR '17, May 2017, pp. 37–41.

[33] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. ACM, 2012, pp. 31–40.

[34] J. Singer, S. E. Sim, and T. C. Lethbridge, "Software engineering data collection for field studies," in *Guide to Advanced Empirical Software Engineering*. Springer London, 2008, pp. 9–34.

[35] S. Mujahid, G. Sierra, R. Abdalkareem, E. Shihab, and W. Shang, "Examining user complaints of wearable apps: A case study on android wear," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 96–99.

[36] H. Zhang and A. Rountev, "Analysis and testing of notifications in android wear applications," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017.

[37] J. Ahola, "Challenges in android wear application development," in *Proceedings of the 15th International Conference on Web Engineering*, ser. ICWE '15. Springer, 2015, pp. 601–604.

[38] K. Lyons, "What can a dumb watch teach a smartwatch?: Informing the design of smartwatches," in *Proceedings of the 2015 ACM International Symposium on Wearable Computers*, ser. UbiComp '15, ACM. ACM, 2015, pp. 3–10.

[39] J. Chauhan, S. Seneviratne, M. A. Kaafar, A. Mahanti, and A. Seneviratne, "Characterization of early smartwatch apps," in *Proceedings of the 2016 IEEE International Conference on Pervasive Computing and Communication Workshops*, ser. PerCom '16. IEEE, 2016, pp. 1–6.

[40] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. ACM, 2010, pp. 73–84.

[41] Android Studio, "Improve your code with lint," https://developer.android.com/studio/write/lint.html, 2017, accessed on June 11, 2017.