Detecting Wearable App Permission Mismatches: A Case Study on Android Wear

Suhaib Mujahid
Data-driven Analysis of Software (DAS) Lab
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
s_mujahi@encs.concordia.ca

ABSTRACT

Wearable devices are becoming increasingly popular. These wearable devices run what is known as wearable apps. Wearable apps are packaged with handheld apps, that must be installed on the accompanying handheld device (e.g., phone).

Given that wearable apps are tightly coupled with the handheld apps, any wearable permission must also be requested in the handheld version of the app on the Android Wear platform. However, in some cases, the wearable apps may request permissions that do not exist in the handheld app, resulting in a permission mismatch, and causing the wearable app to error or crash. In this paper, we propose a technique to detect wear app permission mismatches. We perform a case study on 2,409 free Android Wear apps and find that 73 released wearable apps suffer from the permission mismatch problem.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; *Software maintenance tools*; Parsers;

KEYWORDS

Android Wear; Permissions; Empirical Study; Wearable

ACM Reference Format:
Suhaib Mujahid. 2017. Detecting Wearable App Permission Mismatches: A Case Study on Android Wear. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 3 pages.
https://doi.org/10.1145/3106237.3121279

1 INTRODUCTION

Mobile apps are playing an increasingly important role in our daily lives. These mobile apps can monitor all types of actions, e.g., our location, contacts, etc. To help protect the users privacy and make sure that apps do not intentionally or unintentionally access data that need not be shared, permissions are used to control what an app can access.

Previous work showed that the management of permissions is complicated for the developers and there exists many misuses even for the most popular permissions [1, 5, 10]. With the recent introduction of wearable devices, the management of permissions has become even more complicated. Furthermore a previous study shows that the official documentation for permissions is incomplete [4]. The Android platform (before Android Wear 2.0) stipulates that any permission requested by the wearable version of an app be also requested in the handheld version of the app. These permissions need to be listed in the manifest file of the (wearable/handheld).

However, in some cases, the permissions are not requested properly, i.e., the wearable version of an app may request permissions that are not requested by the handheld version of the app [3]. We call this the permission mismatch problem. The permission mismatch issue is particularly problematic since 1) it does not raise an compilation errors, 2) it does not log any message in the logcat, 3) it runs normally in the emulator or on any wearable device using Android Debug Bridge (adb) and 4) is not automatically detected as a problem by most IDEs, including Android Studio. Often, the permission mismatch problem may lead to any of the following: 1) cause the wearable app to not be installed on the wearable device, 2) cause parts of the wearable app's functionality to error or fail and/or 3) throw a security exception and/or crash the app. To make matters even worse, even under the new permission model, introduced in android API level 23 (requesting permissions at run time) makes the detection and debugging of problems due to permission mismatches even harder. Hence, the permission mismatch problem is usually caught by the user and often leads to a negative user experience, which results in low ratings and revenues.

A wearable app can be categorized as one of the following: 1) a completely independent app, that can be installed directly on the wearable device without any need of a handheld device, 2) a semi-independent app (a handheld app is not required and would provide only optional features) or 3) a dependent wear app, that needs an accompanying handheld app to fully functions [3]. Our technique applies to semi-dependent and dependent wear apps, since they are most are most prone to the permissions mismatch issue.

To help developers avoid releasing wearable apps that suffer from the permission mismatch problem, we develop a technique that analyzes the manifest files of the wearable and handheld versions of an app to ensure that all permissions that are requested by the wearable app are also requested by the handheld app. We perform a preliminary study on the 2,409 free wearable apps from the Google

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3121279

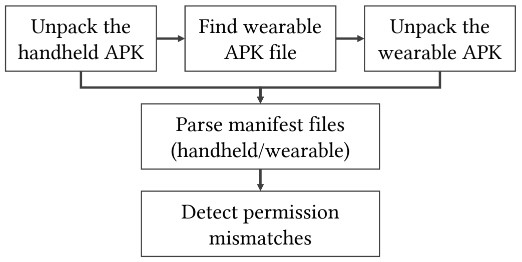
Unpack the handheld APK
Find wearable APK file
Unpack the wearable APK
Parse manifest files (handheld/wearable)
Detect permission mismatches

Figure 1: Overview of the permission mismatches detection process.

Play store and find 73 released wearable apps that contain existing permissions mismatch problems.

The remainder of this paper is organized as follows: Section 2 shows the related work. Section 3 describes our approach to detect the permission mismatch problem. In Section 4, we show the preliminary findings of our study. We conclude the paper and sketch future work in Section 5

2 RELATED WORK

In a previous work [8], we studied the user complaints of wearable apps by analyzing 589 reviews from 6 Android wearable apps. One of our findings indicates that users complain mostly about functional errors of wearable apps. A number of prior studies focused on permission issues in Android apps. For example, some studies found issues and misuses in declaring app permissions [1, 2, 10]. More recently, a number of studies propose techniques to that provide API to permission mappings, in order to mitigate the missed permissions [1, 6]. Jha *et al.* [5] study mistakes in writing Android manifests for mobile apps and they find that more than 78% of studied apps have at least one configuration error. Our study differs from the prior work since we focus on the inconsistent permission problem that may exist between wearable and handheld versions.

3 APPROACH

We describe the overview of our approach in Figure 1. Each part of the approach is detailed in the following subsections.

3.1 Unpack App APKs and Extract Handheld and Wearable Mainfest Files

In order to read the manifest files we unpack the handheld app's APK and decode the resources[1] using Apktool [11]. After obtaining the unpacked resources for the handheld apps, we need to identify the path to the wearable version's APK file, so we apply the following steps: 1) extract the `AndroidManifest.xml` file from the main directory, 2) parse the XML tree of the manifest file, 3) select the meta data tag that refers to the wearable app description file[2] by targeting the tag name `com.google.android.wearable.beta.app`, and 4) parse the XML tree for wearable app description file and extract the path of the wearable APK by targeting the `rawPathResId` tag. A configuration mistake, e.g., a missed declaration of the wearable app description file path, or incorrect APK path could cause a failure in detecting the wearable APK. In this 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, like 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 find the paths of all files with *.apk* extension from the `MANIFEST.MF` file. In case of multiple APK files, we extract and unpack them to figure out which one belongs to the wearable app. We distinguish the wear apps' APKs based on several heuristics, which include: 1) matching the package ID name with the handheld package ID name, 2) looking at the name of the APK file seeking for keywords like *'wear'*, or 3) looking for the usage of tags that indicate the use of wearble hardware in the manifest file, e.g., `android.hardware.type.watch`.

3.2 Detect Missing Permissions

To detect permission mismatches, we extract all the permissions declared in the wearable and handheld manifest files by targeting the `android:name` attribute in all `uses-permission` tags. Afterwards, we compare the wearable and handheld permission lists and detect any permission for the wearable version that does not exist in the manifest file of the handheld version.

However, we exclude the permissions from protection level `normal`. This protection level applies lower-risk permissions that allow requesting applications to access isolated application-level features, with minimal risk to other applications, the system, or the user. The system automatically grants these type of permissions to a requesting application at installation time, without asking for the user's explicit approval (though the user always has the option to review these permissions before installing). Since these permission do not require the explicit user approval, it is allowed to declare them in the wearable manifest without listing them in the handheld manifest file.

Additionally, we ignore the permissions that have been already deleted from the Android wear platform and do not have an effect on the functionality of the app, e.g., `PROVIDE_BACKGROUND`, which can be safely removed.

4 PRELIMINARY FINDINGS

We select the available Android Wear apps on Google Play Store by collecting their identifiers from two alternative app markets: *Android Wear Center* [9] and GoKo [7]. By filtering paid apps from the set of 4,722 apps, we end up with 2,417 free wearable app. 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 developed a crawler that interfaces with the Google play store API as a regular mobile device.

Using the approach described in Section 3, we were able to analyze the permissions of 2,409 apps. We find that 73 of the examined apps suffer from the permission mismatch problem. Of the 73 apps, the number of missed permissions ranges between 1 to 4 permissions, with a median of one missed permission per app. We also investigate which permissions are missed. Table 1 shows the missed permission types and the number of cases for each one of them.

[1]which includes the reverse engineered source code, etc.
[2]A file that contains the version and path information of the wearable app APK.

Table 1: List of missed permissions and the number of apps the permissions are missed. *Others* present all permission types that appears just one time.

Permissions	Count
READ_CALENDAR	10
READ_PHONE_STATE	9
WAKE_LOCK	9
ACCESS_FINE_LOCATION	8
READ_EXTERNAL_STORAGE	7
WRITE_EXTERNAL_STORAGE	7
BODY_SENSORS	5
VIBRATE	5
ACCESS_COARSE_LOCATION	2
RECORD_AUDIO	2
SYSTEM_ALERT_WINDOW	2
Others	28

We see that the most commonly missed permissions are related to calendar, phone state and wake/lock.

Our findings show that the permission mismatch problem does exist in wearable apps and that certain permissions are more likely to be missed than others. This motivates us to study these mismatch problems further and to develop techniques to help developers avoid such issues.

5 CONCLUSION AND FUTURE WORK

Wearable device popularity is increasing. In fact, based on our data collection, Google Play Store contains more than 4,700 wearable apps. One of the requirements to properly package a semi- or dependent wearable app is to include all the permissions declared in the manifest file of the wearable app in the manifest file of the handheld app. Our study shows that the permissions mismatch problem exists in 73 wearable apps and that can negatively impact the quality of the apps. In the future, we plan to further build techniques to help developers deal with permission inconsistency problems.

REFERENCES

[1] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, 217–228.

[2] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. 2010. 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 (CCS '10)*. ACM, 73–84.

[3] Android Developers Documentation. 2017. Packaging Wearable Apps. https://developer.android.com/training/wearables/apps/packaging.html. (2017). (Accessed on 01/19/2017).

[4] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, 627–638.

[5] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. 2017. Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, 25–36.

[6] M. Y. Karim, H. Kagdi, and M. D. Penta. 2016. Mining Android Apps to Recommend Permissions. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 427–437.

[7] Jakob KÃűrner, Lars Hitzges, and Dennis Gehrke. 2016. Goko. http://goko.me. (2016). (Accessed on 09/09/2016).

[8] Suhaib Mujahid, Giancarlo Sierra, Rabe Abdalkareem, Emad Shihab, and Weiyi Shang. 2017. 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*. IEEE Press, 96–99.

[9] Wearable Software. 2016. Android Wear Center. http://www.androidwearcenter.com. (2016). (Accessed on 09/09/2016).

[10] R. Stevens, J. Ganz, V. Filkov, P. Devanbu, and H. Chen. 2013. Asking for (and about) permissions used by Android apps. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 31–40.

[11] Connor Tumbleson and Ryszard WiÅŻniewski. 2017. Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/. (2017). (Accessed on 05/04/2017).