Determining and Detecting Permission Issues of Wearable Apps

Suhaib Mujahid

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

April 2018

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Suhaib Mujahid**

Entitled: **Determining and Detecting Permission Issues of Wearable Apps**

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

Master of Applied Science (Software Engineering)

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

Signed by the Final Examining Committee:

_____ Chair
Dr. Weiyi Shang

_____ Examiner
Dr. Yann-Gael Gueheneuc

_____ Examiner
Dr. Tse-Hsun Chen

_____ Supervisor
Dr. Emad Shihab

Approved by _____
Sudhir Mudur, Chair
Department of Computer Science and Software Engineering

_____ 2018

_____
Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Determining and Detecting Permission Issues of Wearable Apps

Suhaib Mujahid

Wearable apps are becoming increasingly popular. Nevertheless, to date, very few studies have examined the issues that wearable apps face. Prior studies showed that user reviews contain a plethora of insights that can be used to understand quality issues and help developers build better quality mobile apps.

Therefore, in this thesis, we start by empirically studying user reviews to understand the user complaints about wearable apps. We manually sample and categorize 2,667 reviews from 19 Android wearable apps. Additionally, we examine the replies posted by developers in response to user complaints. This study allows us to determine the type of complaints that developers care about the most and to identify problems that, despite being important to users, do not receive a proper response from developers.

We find that the most frequent complaints are related to Functional Errors, Cost, and Lack of Functionality, whereas the most negatively impacting complaints are related to Installation Problems, Device Compatibility, and Privacy & Ethical Issues. We find that developers mostly reply to complaints related to Privacy & Ethical Issues, Performance Issues, and notification-related issues. Furthermore, we observe that when developers reply, they tend to provide a solution, request more details, or let the user know that they are working on a solution. Our results highlight the issues that users face the most, and the issues to which developers should pay additional attention to due to their negative impact.

Based on these results from the first empirical study, we investigate the most negatively impactful complaints. We observe that mainly two permission problems are a common factor to raise

issues that cause these complaints -namely the permission mismatch problem and the problem of superfluous features. As a result, we propose a technique to detect permission problems in wearable app. To operationalize our technique we developed a tool, called PERMLYZER, that automatically detects these two problems from Android APKs. We then perform an empirical study on of 2,724 free wearable apps. Our findings show that the permission mismatches exist in 6.1% of released apps on the app store. Moreover, we find that 19.2% of studded wearable apps contain superfluous features.

Related Publications

The following publications are related to this thesis:

- **S. Mujahid**, G. Sierra, R. Abdalkareem, E. Shihab, W. Shang, Examining User Complaints of Wearable Apps: A Case Study on Android Wear, In *Proceedings of 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'17)*.

- **S. Mujahid**, Detecting Wearable App Permission Mismatches: A Case Study on Android Wear, In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'17)*, Student Research Competition.

- **S. Mujahid**, G. Sierra, R. Abdalkareem, E. Shihab, W. Shang, An Empirical Study of Android Wear User Complaints, *In Empirical Software Engineering Journal (EMSE)*, 2018. [Major Revision]

The following publications are not directly related to the material in this thesis, but were produced in parallel to the research performed for this thesis.

- R. Abdalkareem, O. Nourry, S. Wehaibi, **S. Mujahid**, and E. Shihab, Why Do Developers Use Trivial Packages? An Empirical Case Study on npm, In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'17)*.

Statement of Originality

I, Suhaib Mujahid, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Dedication

To my parents, my sisters, and my grandmother.

Acknowledgments

First and foremost, I would like to thank Allah for blessing me with this opportunity.

I would like to thank my supervisor Dr. Emad Shihab for his endless support and dedication during this journey, as well as his unparalleled motivation and immense knowledge, without which this thesis would not have been completed. I have learned a great deal from you, words could never be enough to express my gratitude. You played formative roles in my development as a researcher and as a person.

My appreciation extends to my fellow colleagues, Rabe Abdalkareem, Giancarlo Sierra, Sultan Wehaibi, Ahmad Abdellatif, Jinfu Chen, Kundi Yao, Moiz Arif, Mohamed Elshafei and everyone else in the Data-driven Analysis of Software (DAS) Lab.

It is a pleasure to thank my friends who inspired and supported me tirelessly through thick and thin. A special thanks to Samer Sioure, Rame Kateb, Saif Masry, Amro Doufish, Khalil Edais, Ehab Ewaiwi and Asem Mujahed. You guys always fueled me with utmost motivation. I am proud to call you my closest friends.

Lastly, I thank my parents and my grandmother for always being there for me. The greatest motivation that makes me continue my path toward success is seeing joy and pride in their faces. I likewise thank my sisters for their unwavering moral support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Problem Statement

Wearable devices i.e., smart watches and fitness trackers, are becoming increasingly popular and are expected to reach 101 million devices by 2020 (Chauhan, Seneviratne, Kaafar, Mahanti, & Seneviratne, 2016). Wearable devices have unique characteristics that pose challenges when compared to other platforms or devices (Rawassizadeh, Price, & Petre, 2015). These devices provide their developers with access to a diverse set of sensors and features (e.g,. physiological, biochemical, as well as motion sensing (Bonato, 2010; Teng, Zhang, Poon, & Bonato, 2008)) that can be used to enhance the user experience (Android documentation, 2016a). As such, developers began to develop apps that are specifically designed to run on these wearable devices, called wearable apps. Wearable apps are different than handheld apps that run on mobile phones (Wright & Keith, 2014), since they: 1) are often very lightweight (resource wise) (Park & Jayaraman, 2003), 2) are meant to run on very small screens (Tehrani & Michael, 2014), 3) have access to a different set of sensors (Do, Martini, & Choo, 2017), and 4) heavily depend on a mobile device to perform most of the expensive processing (Chauhan et al., 2016; J. Wei, 2014).

A fundamental change introduced by mobile apps (i.,g. handheld and wearable apps) is the way that they are released to users, which is through app stores. App stores allow users to directly provide feedback on the mobile apps through user reviews. Although these user reviews were meant to simply provide feedback about the apps, they proved to be much more useful (Galvis Carreño & Winbladh, 2013; Harman, Jia, & Zhang, 2012; Pagano & Maalej, 2013). For example, studies have shown that they can be used to prioritize devices to test (Khalid, Nagappan, Shihab, & Hassan,

2014), prioritize feature improvements (Keertipati, Savarimuthu, & Licorish, 2016), and/or can be used to understand user problems so that developers can avoid low ratings, which can have a major impact on the app's user base, revenues and the success of the app in general (Di Sorbo et al., 2016; Finkelstein et al., 2017; Guzman & Maalej, 2014).

Because wearable apps are a new paradigm, we leveraged these user reviews to understand the problems users face when using wearable apps so that developers and researchers can address them. We perform a qualitative study, involving more than 2,600 user reviews and present our findings in Chapter 3 of this thesis. One of our main findings is that connection and device compatibility issues are some of the most impacting issues for users of wearable apps. Hence, we next perform an empirical study to investigate the source of these highly negatively impacting issues. We find that the permissions model used for wearable apps is one source of these problems, that can be automatically fixed. Hence, in Chapter 4 of the thesis, we empirically analyzed the permissions issues that arise. Our thesis contributions help build solid empirical knowledge in a new field, wearable apps.

In the rest of the introduction, we provide a more detailed overview of the thesis' chapters and list our contributions.

1.1 Thesis Overview

The body of the thesis is composed of two main chapters:

Chapter 3: An Empirical Study of Android Wear User Complaints

Since the practice of developing wearable apps is a relatively new concept, we start by understanding the main issues in developing these wearable apps. To do so, we study user complaints through the examination of users reviews posted on Google Play store. We do so since previous studies showed that user review are a rich source of information that can highlight the main problems faced by app users and reflect the challenges of developing such apps (Palomba et al., 2015; Panichella et al., 2015). We conducted a qualitative analysis where we analyze 2,667 user's review and find that:

- *Functional Errors*, *Cost*, and *Lack of Functionality* are the three most frequent complaints.
- *Installation Problems*, *Device Compatibility*, and *Privacy & Ethical Issues* are the most negatively perceived by users. Users that encounter *Installation Problems* of wearable apps are five times more likely to give a 1-star review than a 2-star review.
- Developers are most likely to reply to complaints related to *Privacy & Ethical Issues*, *Performance Issues*, and *Spam Notifications*. We also contrast the complaints based on their impact and the developer replies and find that *Installation Problems*, *Device Compatibility*, and *Connection & Sync Issues* are most impacting, but have a low response rate from developers.
- When developers reply to user complaints, they often try to get more information or provide potential workarounds to solve the complaints.

Chapter 4: Detecting Permission Problems of Wearable Apps

Motivated by the finding in the previous study, particularly the fact that most impacting issues are related to installation and compatibility issues, we further empirically examined issues related to permissions in the context of wearable apps. We identified two main permission problems: 1) permission mismatch, which refers to the case where the permissions declared in the mobile and the wearable versions of the app do not match - such permission mismatches may cause installation and connection issues; and 2) superfluous features, which refers to the case where an app requests a permission that requires a specific hardware resource (e.g., access to the microphone) without really needing that permission - such superfluous features can cause device incompatibility issues. We conducted an empirical study on 2,724 wearable apps to examine the prevalence of these problems and their impact on the wearable apps. Our findings show that:

- The permission mismatch problem exists in 6.1% currently *released* apps on the Google Play store.
- Out of the apps that requires underlying (hardware) features, 52.4% of embedded wearable apps and 80.5% of standalone wearable apps have at least one superfluous feature.

- 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.

To alleviate the problems due to permission mismatches and superfluous features, we devise a technique, called PERMLYZER that can help automatically highlight and suggest fixes for the two aforementioned problems.

1.2 Thesis Contributions

The main contributions of the thesis are:

- Empirically investigate user complaints in wearable apps by manually classify 2,667 reviews belonging to 19 wearable apps; then measured the frequency of the complaints and how negatively they are perceived by users.
- Examine the developer replies to the studied complaints in order to better understand the areas that receive enough attention and areas that are important to the users.
- Define and examine the two main permission problems that are related to the most negatively impactful user complaints; then perform an empirical study to examine them through investigating 2,724 embedded apps and 339 standalone wearable apps.
- Implement our approach of detecting wearable permissions' problems in a tool called PERMLYZER, which will be freely available.
- Provide our dataset of the crawled apps' APKs, the detailed analyses data, the collected 1.2 million reviews, and the manually classifying of the reviews to be publicly available.

Chapter 2

Literature Review

The work that is most related to our study falls into three main categories: work that leveraged mobile user reviews, work focusing on wearable apps, and work focusing on Android permissions.

2.1 Work Leveraging Mobile User Reviews

One of the first studies to leverage mobile app reviews was done by Harman et al. (2012). In their paper, the authors studied the correlation of user reviews with key performance metrics such as the number of downloads. They found that there is a strong correlation between app ratings and its rank based on the number of downloads, suggesting that developers should pay close attention to their user ratings. More recently, Finkelstein et al. (2017) extended the work by Harman et al. (2012), which mined data from the Blackberry World App Store to analyze the correlation between: the customer rating of an app from user reviews, its price, popularity (based on downloads), and claimed features that extracted from each app's description with natural language processing (NLP) techniques. The authors found that there is a strong correlation between the customer rating of an app and its popularity, and a moderate correlation between price and the claimed features of an app.

Other studies mined user reviews to better understand the contents of these user reviews. Khalid et al. (2015) studied low-rated user reviews from 20 free iOS apps in order to help developers understand their nature. They exposed 12 types of complaints and found that feature requests, functional errors and, crashing apps were the most frequent reasons for negative reviews, while

privacy and ethical concerns corresponded to the most impactful reviews that mostly lowered the rating of an app. Ha and Wagner (2013) manually analyzed the user reviews of 59 Android apps to examine the impact of privacy and ethical issues. They found that only around 1% of the apps contain complaints related to privacy and ethical issues. Hoon, Vasa, Schneider, and Mouzakis (2012) and Vasa, Hoon, Mouzakis, and Noguchi (2012), reviewed the vocabulary of 8.7 million user reviews from the Apple App Store showing a link between the length of a review and its given rating.

In other work, Fu et al. (2013) automated the analysis of over 13 million reviews of more than a hundred thousand apps in the Google Play store using Latent Dirichlet Allocation model (LDA). They uncovered 10 unique topics corresponding to user complaints; they also found that there is a significant difference between free and paid apps because paid apps often present a complaint topic of the involved pricing, absent in the user reviews of free ones. Similarly, Chen, Lin, Hoi, Xiao, and Zhang (2014) created a framework to automatically extract the most informative reviews from a data set of mobile apps using NLP techniques. They found that frequently, the amount of reviews for an app can be too large for human reading or understanding, and that only 35.1% of the reviews actually contain valuable information that developers could use for app improvement. Therefore, their framework automates an approach to filter, group, rank and visualize the informative portions of the reviews only.

Other work by McIlroy, Ali, Khalid, and Hassan (2016) found that up to 30% of mobile app reviews can contain multiple topics of information and proposed an automated approach for labeling the user reviews, which reached a precision of 66% and 65% of recall while classifying them in 13 different categories.

Regarding categorization of developer replies to the low scored user reviews, a recent study by McIlroy, Shang, Ali, and Hassan (2015) introduced the benefits of responding to app reviews, indicating that following a response users would increase the review rating 38.7% of the time by 20% of the previous score.

Panichella et al. (2015) studied the structure, sentiment and text features of mobile app reviews and proposed: 1) a taxonomy of 4 categories related to software maintenance and evolution tasks in which to classify app user reviews; and 2) an approach to automatically classify them using

NLP, text analysis and sentiment analysis techniques. The authors combined these techniques using machine learning and empirically evaluated their classifiers, showing that their approach can aid developers to obtain the intention from user reviews. Later, Panichella et al. (2015) extended their work and implemented their approach in a tool named ARDOC Panichella et al. (2016) that automates the classification of user reviews. The performance of the tool was validated by the developers of 3 real-world mobile apps and an external software engineer. ARDOC achieved promising results with precision, recall and F-Measure values ranging between 84% to 89%.

Di Sorbo et al. (2016) introduced a model to obtain the topics contained in user reviews from mobile apps, which they call URM (User Reviews Model). The model was combined with the approach presented in Panichella et al. (2015) to capture the intentions of user reviews in a new approach named SURF (Summarizer of user reviews Feedback). SURF generates summaries from sets of user reviews and clusters them considering both, the intention and topics found in user reviews to recommend software changes. The usefulness of this approach was validated on 17 mobile apps by 23 developers and researchers. As a follow-up, Di Sorbo, Panichella, Alexandru, Visaggio, and Canfora (2017) implemented and validated SURF as a tool to automate the processing of user reviews for developers.

More recently, Ciurumelea, Schaufelbhl, Panichella, and Gall (2017) manually analyzed 1,566 user reviews from 39 mobile apps and defined a multi-level taxonomy that is specific to the mobile domain. The authors introduced an approach, called URR (User Request Referencer) that not only automatically classifies user reviews in their multi-level taxonomy, but also points developers to the artifacts that need to be modified to address a particular user review. They showed that doing so reduces the time it takes to process user reviews manually by up to 75%.

With another perspective, Palomba et al. (2017) presented CHANGEADVISOR, an approach that clusters multiple user reviews that contain change requests to recommend developers which artifacts to modify in a mobile app to address user feedback. This approach uses NLP and clustering techniques to sort reviews based on their content, semantics and structure. A validation conducted with the developers of 10 mobile apps highlighted the usefulness of this approach when mining large numbers of user reviews, providing 81% of precision and 70% recall when recommending changes.

There are also a plethora of other works on mobile apps, that leverage users reviews for their techniques. Due to space limitations, we only discuss the most relevant studies in this section, however, we refer the reader to a recent survey by Martin, Sarro, Jia, Zhang, and Harman (2017) for a more comprehensive list of studies on mobile apps.

Our work differs from the prior work since 1) we focus on the user reviews of wearable apps, 2) we triangulate two data sources user reviews and developers replies to understand the types of user complaints that developers care about.

2.2 Work Focusing on Wearable Apps

Very few studies have focused on the study of wearable apps, but many different paths are beginning to get explored in the domain. In our previous work (Mujahid, Sierra, Abdalkareem, Shihab, & Shang, 2017), we studied the user complaints of wearable apps by analyzing 589 reviews from 6 Android wearable apps. Our main findings indicate that users complain most about functional errors, lack of functionality, and cost of wearable apps.

Recently, Zhang and Rountev (2017) presented a 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 (2015) exposed 3 issues and limitations in Android Wear platform found 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 (2015) did a study on the user perceptions of functionality and design of smart watches, including android wearable devices. Based on user feedback and in contrast to traditional watches, possible features for future wearable app are suggested. Min et al. (2015) explored the battery usage of wearable apps and performed an online survey to get direct feedback and concerns from users. They found that most users do not complain about the battery usage of their wearable devices. Chauhan et al. (2016) did a previous categorization of smart watch apps from Samsung, Apple, and Android Wear. They used data from *Android Wear Center* (Wearable Software, 2016) and *GoKo* (Korner, Hitzges, & Gehrke, 2016) as sources to get the wearable app identifiers for crawling their information; we applied the same

approach to initialize our crawling phase.

Our work differs from prior work on wearable apps since, to the best of our knowledge, our work is one of the first to analyze their user complaints in depth. Moreover, we differ from previous work since we not only investigate the categories, frequency and impact of low rated user reviews; but we also contrast our findings to similar ones in the domain of handheld mobile devices. By doing this comparison, we are able to bring several implications for wearable app development into the community spotlight.

2.3 Work Focusing on Manifest and Permissions

A number of studies focused on permission issues in the context of Android apps. For example, Stevens, Ganz, Filkov, Devanbu, and Chen (2013), and Barrera, Kayacik, van Oorschot, and Somayaji (2010) found issues and misuses in declaring app permissions. Barrera et al. (2010) performed an empirical analysis on the expressiveness of Android's permission sets and discussed some potential improvements for the Android permission model. Also, their study found that while Android has a large number of permissions restricting access to advanced functionality on devices, only a small number of these permissions are actively used by developers.

X. Wei, Gomez, Neamtiu, and Faloutsos (2012) conducted a study on the evolution of Android permissions, focusing how the set of permissions has evolved. They analyzed patterns and permission distributions, and reported that applications tend to be overprivileged and to request more permissions over time.

Felt, Chin, Hanna, Song, and Wagner (2011) proposed the first solution to automatically detect overprivileged permissions in Android apps and found 33.7% of the apps in their study were overprivileged. Their tool STOWAWAY depends on automated testing tools to analyze the Android APIs in order to build the permission map that is necessary for detecting overprivileged permissions. Au, Zhou, Huang, and Lie (2012) present the tool PSCOUT that outperform STOWAWAY which adopted static analysis on the source code of the Android platform to extract the permission map. Later, Pandita, Xiao, Yang, Enck, and Xie (2013) present WHYPER, a framework using Natural Language Processing (NLP) techniques to identify sentences that describe the need for a given permission

in an app description. More recently, Karim, Kagdi, and Penta (2016) presents APMINER, which recommend the permissions that the app should request based on the usage of APIs and permissions in other apps published in the app store. Bao, Lo, Xia, and Li (2016) propose an approach to o improve the effectiveness of a prior work by Karim et al. (2016). The proposed approach is based on collaborative filtering technique, which considers apps that use similar APIs, usually support similar features, so the required permissions are usually similar too.

Jha, Lee, and Lee (2017) built a tool called MANIFESTINSPECTOR to detect mistakes in writing Android manifests for mobile apps (including the permissions). They found that more than 78% of the studied apps have at least one configuration error. Android LINT (Android Studio, 2017b) is an analysis tool built by Google that statically analyzes source code files, including manifest files. MANIFESTINSPECTOR (Jha et al., 2017) is one of the best performing tools when it comes to number of 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. Nevertheless, none of these rules specifically target the manifest files' configuration for wearable apps.

Our study differs from prior work since we focus on the inconsistent configuration problems (permission mismatches and superfluous features) that may exist between wearable and handheld versions of an app.

Chapter 3

An Empirical Study of Android Wear User Complaints

3.1 Introduction

Mobile apps are very popular and have been the focus of numerous studies in recent years (Martin et al., 2017; Nagappan & Shihab, 2016). A fundamental change introduced by mobile apps is the way that they are released to users, which is through app stores. App stores allow users to directly provide feedback on the mobile apps through user reviews. Although these user reviews were meant to simply provide feedback about the apps, they proved to be much more useful (Galvis Carreño & Winbladh, 2013; Harman et al., 2012; Pagano & Maalej, 2013). For example, studies have shown that they can be used to prioritize devices to test (Khalid et al., 2014), prioritize feature improvements (Keertipati et al., 2016), and/or can be used to understand user problems so that developers can avoid low ratings, which can have a major impact on the app's user base, revenues and the success of the app in general (Di Sorbo et al., 2016; Finkelstein et al., 2017; Guzman & Maalej, 2014).

Recently, wearables that complement handheld devices were introduced. Wearable devices i.e., smart watches and fitness trackers, are becoming increasingly popular and are expected to reach 101 million devices by 2020 (Chauhan et al., 2016). Wearable devices have unique characteristics that pose challenges when compared to other platforms or devices (Rawassizadeh et al., 2015).

These devices provide their developers with access to a diverse set of sensors and features (e.g,. physiological, biochemical, as well as motion sensing (Bonato, 2010; Teng et al., 2008)) that can be used to enhance the user experience (Android documentation, 2016a). As such, developers began to develop apps that are specifically designed to run on these wearable devices, called wearable apps. Wearable apps are different than handheld apps that run on mobile phones (Wright & Keith, 2014), since they: 1) are often very lightweight (resource wise) (Park & Jayaraman, 2003), 2) are meant to run on very small screens (Tehrani & Michael, 2014), 3) have access to a different set of sensors (Do et al., 2017), and 4) heavily depend on a mobile device to perform most of the expensive processing (Chauhan et al., 2016; J. Wei, 2014). To the best of our knowledge, very few studies have focused on wearable apps and their user reviews to this date.

Therefore, similar to the prior studies on (handheld) mobile app reviews (Ha & Wagner, 2013; Hoon et al., 2012; Khalid et al., 2015; Vasa et al., 2012), we also investigate user complaints but focusing on reviews from wearable apps. Note that the goal of our study is not to highlight or surface differences between complaint types from mobile and wearable apps. We aim to investigate user complaints in wearable apps (even if they are the same than for mobile). With this in mind, we did find some complaint types differ from the ones found in the mobile domain.

To perform our study, we manually classify 2,667 reviews belonging to 19 wearable apps. The reviews were tagged by two independent researchers and grouped into 15 different categories. For each category, we measured the frequency of the complaints and how negatively they are perceived by users. We measure this negative perception based on how low they rate complaints of a certain category. Since this negative perception is reflected into low user ratings, we rank the impact of each complaint category based on the ratio of 1-star rated reviews to 2-star rated reviews.

We also examine the developer replies to these complaints in order to better understand the areas that receive enough attention and areas that are important to the users, but not well attended by the developers. Our study concerns two main areas: I) examining user complaints and II) examining developer replies. For each area, we ask two research questions:

I.1 What do Wearable App Users Complain About?

Our findings indicate that *Functional Errors*, *Cost*, and *Lack of Functionality* are the three

most frequent complaints.

I.2 What User Complaints are Most Negatively Impacting?

We find that *Installation Problems*, *Device Compatibility*, and *Privacy & Ethical Issues* are the most negatively perceived by users. Users that encounter *Installation Problems* of wearable apps are five timesmore likely to give a 1-star review than a 2-star review.

Our findings provide insight to the developer and research community as to what issues wearable app users face the most and which issues are most impactful.

II.1 What Types of Complaints Do Developers Reply To?

In addition, we also examined the developer replies to the user complaints. We find that developers are most likely to reply to complaints related to *Privacy & Ethical Issues*, *Performance Issues*, and *Spam Notifications*. We also contrast the complaints based on their impact and the developer replies and find that *Installation Problems*, *Device Compatibility*, and *Connection & Sync Issues* are most impacting, but have a low response rate from developers.

II.2 How Do Developers Reply to Complaints?

We find that when developers reply to user complaints, they often try to get more information or provide potential workarounds to solve the complaints.

Our results highlight areas that are of high importance to the users, but are not well addressed by the developers, and vice versa.

In addition, we compare our findings to the handheld user complains reported by Khalid et al. (2015). Our findings show that 11 of the 15 categories found in our research are common to both handheld and wearable apps, however, 4 of the complaint types are unique to wearable apps, namely - *Lack of Functionality*, *Connection & Sync*, *Spam Notifications*, and *Missing Notifications*. Moreover, we find that similar to their findings, approximately 12% of the complaints occur after an update. Our findings show that there is a need to ensure regression tests are performed before wearable apps are updated. Furthermore, to enable future research and enable the replication of this work, we make our dataset publicly available [1].

[1] https://github.com/suhaibtamimi/user_complaints_of_wearable_apps_dataset

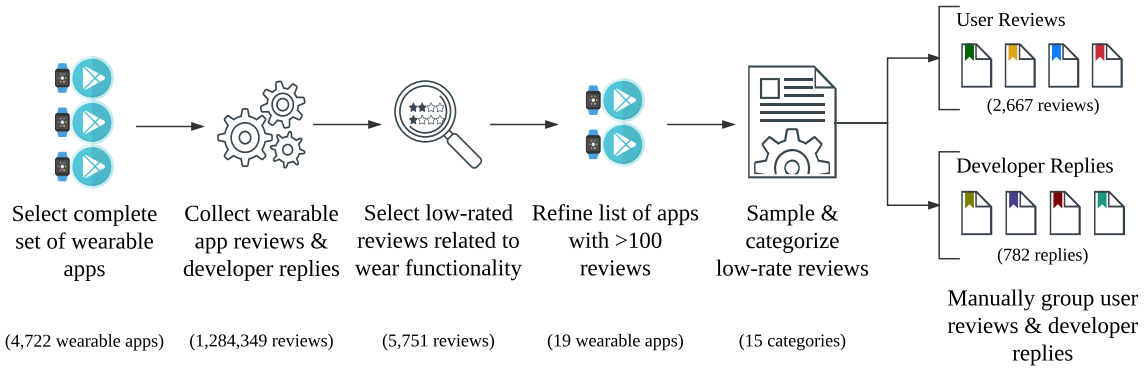


Figure 3.1: Overview of the User Review and Developer Reply Classification Process

The rest of the chapter is organized as follows. Section 3.2 details our study design, including our collection and selection methodology. Section 3.3 presents our results. Section 3.4 compares the findings for wearable and handheld devices. Section 3.5 discusses possible the threats to validity. Section 3.6 concludes the chapter.

3.2 Study Design

The goal of our study is: 1) to determine the most frequent and negatively impacting user complaints of wearable apps and 2) to investigate the type of complaints that developers reply to and the reply types. To do so, we mine the Google Play store for the reviews of wear apps. Figure 3.1 provides an overview of our approach. In the following sections we describe our data selection and collection, as well as detail our review classification methodology.

3.2.1 Data Collection and Selection

For the purpose of our study, we select a number of wearable apps that have user reviews. First, we obtained the available Android Wear apps on Google Play store by collecting their identifiers from two alternative app markets: *Android Wear Center* (Wearable Software, 2016) and *GoKo* (Korner et al., 2016) that accessed on September 9th, 2016. The two aforementioned sources have been used in prior work focusing on wearable apps (Chauhan et al., 2016). Then, we mined the wearable apps using a data scrapper that we wrote. The scrapper collected various information about each wear apps, including: the user reviews' text, its rating, the developer's reply to the review, if any,

and the apps' overall rating. To enhance performance of the scrapper, it was deployed on a cluster of machines in order to distribute the requests sent.

In total, we mined the data of 4,722 wearable apps from 2,732 unique developers, which contained 1,284,349 user reviews. From the total number of mined apps, we found that 1,017 app did not contain any user review at all, i.e., 21.5% of apps. Since we are interested in user complaints, we selected low-rated reviews only (i.e., 1 and 2 stars rating). This was done following a prior study by Khalid et al. (2015), with the rationale that low-rated reviews are most likely to contain user complaints. We also noted, that 1,958 apps did not contain any 1 or 2 star rated user reviews, i.e., 41.5% of apps. Considering we need a reasonable amount of data to perform our analysis, we only selected apps with over 100 low-rated reviews. This left us with 5,751 low-rated user reviews from 19 wearable apps. Note that we include data from all available releases (up to the collection date) of the 19 wearable apps.

Since this is the first study to examine user complaints for wearable apps (in addition to our preliminary short study), we opt to perform our analysis of the user complaints manually. Given that this manual classification is a time and resource intensive task, we selected a random statistically representative sample of complaints from each application. The sample sizes were selected to attain a 5% confidence interval and a 95% confidence level in the population being sampled. This random sampling process resulted in 2,667 total reviews varying from 89 to 307 reviews per app. The list of the studied wearable apps, their category on Google Play store, cost, overall rating, the number of examined reviews, number of developers' replies, and data span of reviews are shown in Table 3.1. This data was collected in October 2016.

3.2.2 Manual Classification of User Reviews

Once we obtained all of the reviews, we took a statistical significant random sample[2] of 597 reviews from all the selected apps. We manually inspected and classified the sampled reviews twice, (by two independent researchers) into different categories using an open coding approach (Seaman, 1999; Usman, Britto, Brstler, & Mendes, 2017).

[2]The random sample of 597 reviews was taken out of 5,751 low-rated reviews to achieve a confidence level of 99% and a confidence interval of 5%.

Table 3.1: Statistics of Studied Android Wearable Apps

Wear App Name	Low Rated Reviews	Sampled Reviews	Developer Replies	Date span of Reviews
ZenWatch Manager	201	132	40	26/11/14 - 06/10/16
WatchMaker Premium Watch Face	501	218	22	31/03/15 - 07/10/16
Odyssey Watch Face	125	94	20	10/12/14 - 21/08/16
Skymaster Pilot Watch Face	152	109	20	02/11/14 - 28/09/16
Ranger Military Watch Face	163	115	23	29/12/14 - 20/09/16
Wear Mini Launcher	133	99	44	21/08/14 - 05/10/16
Wear Face Collection	136	101	21	18/07/14 - 20/09/16
InstaWeather for Android Wear	141	103	61	18/12/14 - 30/09/16
Motorola Connect	213	137	20	07/09/14 - 06/10/16
Watch Faces for Android Wear	154	110	10	25/10/14 - 30/09/16
Facer Watch Faces Android Wear	926	272	134	01/08/14 - 07/10/16
Bits Watch Face	116	89	59	20/08/15 - 01/10/16
WatchMaster - Watch Face	124	94	85	24/07/15 - 12/10/16
Luxury Watch Faces for Wear	115	89	79	02/09/14 - 07/10/16
Android Wear - Smartwatch	1,531	307	79	29/09/15 - 08/10/16
Weather Watch Face	279	162	32	20/07/14 - 21/09/16
Web Browser for Android Wear	142	104	33	23/07/14 - 04/10/16
Plants vs. Zombies Watch Face	369	188	0	03/01/15 - 30/09/16
LG Call for Android Wear	230	144	0	28/04/15 - 19/09/16
Total	**5,751**	**2,667**	**782**	**-**

Both classifications were done individually and independently. Each of the two classifiers classified the review into a certain category based on its content. Disagreements between the two classifiers were clarified. This step was done mainly to come up with an initial set of categories that the reviews can be grouped into. For both researchers the categories were defined by the first half of the sampled reviews. By the end of this step, the two researchers defined 15 different initial categories. Note that throughout the thesis we also refer to these categories as complaint types.

Once we defined the initial 15 complaint types, we proceeded to categorize our set of reviews composed by samples of each studied wear app (in total 2,667 reviews). To facilitate the categorization of the reviews, we built a web-based tool. Figure 3.2 shows the main page of the developed tool. It presented for each of the two people categorizing the review with all the review details and the respective developer reply, if a developer posted a reply to the review. The tool also had the option to add a new category in case a review belonged to a category that was not in our initial set.

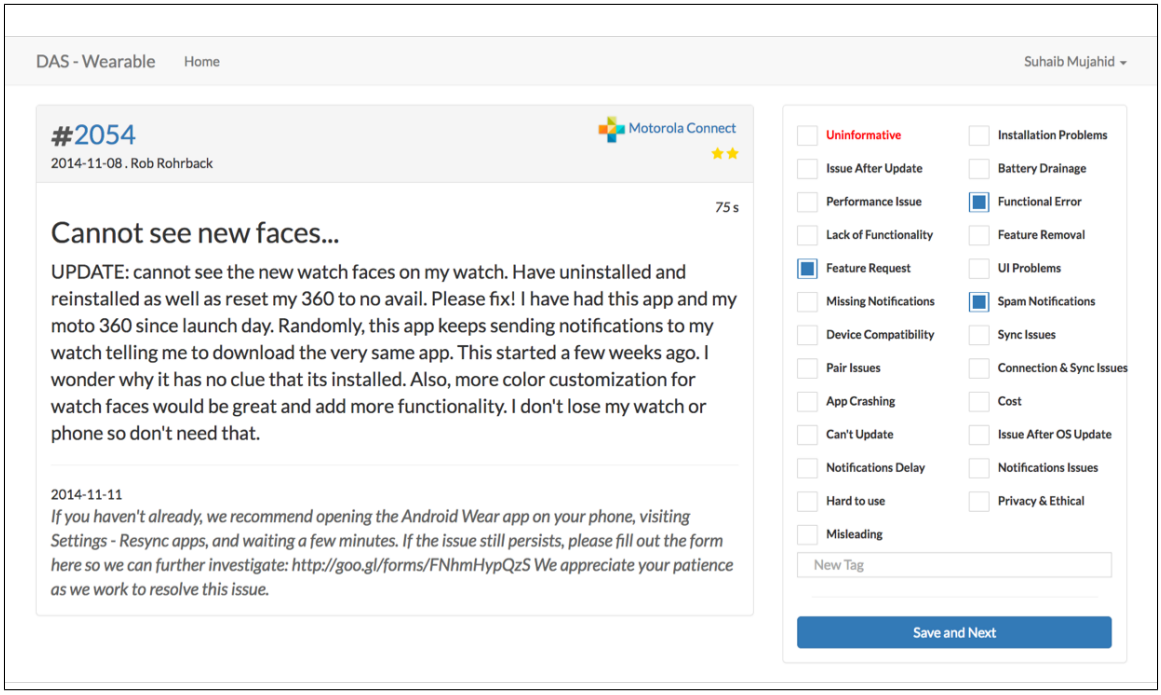


Figure 3.2: Web application for classifying wearable apps' user reviews.

However, even though the tool had the option to add a new category in case a review belonged to a category that was not in our initial set, the researchers did not come up with any new categories. Every review was tagged with all suitable categories, i.e., one review can have one or multiple tags based on its content. For example: if a user complaint mentions a battery drainage problem and also a connection issue, the review will be classified with the *Connection & Sync Issues* and *Battery Drainage* tags. In some instances, the user provided uninformative content in his review (e.g., "Just nonsense, I hated this game..."), in which case we put them in the 'Uninformative' category. The process to categorize the user reviews took approximately 115 hours in total.

As in any other human activity, there may be some disagreements when classifying the user reviews, and therefore, we applied a *Cohen's Kappa* to measure the level of agreement between the two individual classifications (Cohen, 1960). The Cohen's Kappa coefficient has been commonly used to evaluate inter-rater agreement level for categorical scales, and provides the proportion of agreement corrected for chance. The resulting coefficient is scaled to range between -1 and +1,

where a negative value means poorer than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement (Fleiss & Cohen, 1973). The closer the value is to +1, the stronger the agreement.

The level of agreement was +0.68, which is considered to be fair to good agreement (Fleiss & Cohen, 1973). Out of the 2,667 classified reviews, 1,429 reviews had *full agreement* (i.e., both classifiers had the same selection while tagging the reviews). The remaining 1,238 reviews had a conflict in the classification, and for 710 reviews of them, the classification was a match for one or more of the categories but different in some other(s). We examined all the reviews with a classification conflict and did a post agreement on the tags for those reviews; for example, if the first classifier tagged the review as *Device Compatibility* and the second one tagged it as *Functional Error*, we put this review for discussion. The two classifying researchers presented a case s to why they classify a review in a certain category and reached agreement; this scenario solved most conflicts. When both researchers could not agree, a third researcher was consulted to break the tie and reach a final classification.

3.3 Results

Once all the reviews in our dataset are categorized into the different complaint types, we proceed to answer our research questions, pertaining to two areas: user complaints and developer replies. In particular, we are interested in knowing what users complain about and what complaints tend to have the most negative impact. As for the developer replies, we examine the complaints that developers reply to and how they reply to them.

3.3.1 What do Wearable App Users Complain About?

Since wearable apps are an emerging trend, our goal is to understand the types of user complaints so that developers can anticipate potential problems and plan their quality assurance efforts accordingly. Similar to prior studies on user complaints for handheld device apps (Khalid et al., 2015), we start by examining the different types of complaints based on the low-rated reviews of wearable apps.

To come up with the different complaint types, we manually categorized the different wearable app reviews as mentioned earlier in Section 3.2.2. We then rank the different complaint types based on their frequency in the examined reviews.

Table 3.2 shows the 15 different complaint types that we discovered from the wearable app reviews. For each category, we provide a brief description, an example review and the row percentage of reviews in each complaint type. It is important to note that one review can present more than one issue, hence, it can be mapped to more than one complain type. Thus the percentage of reviews sum to more than 100%. From the table, we observe that many of the complaint types are related to the features provided by the wearable apps (e.g., *Feature Removal*, *Feature Request*), the behavior of the wearable apps (e.g., *App Crashing*, *Notifications*, *Battery Drainage*) and external factors (e.g., the *Cost* of the app, *Privacy & Ethical Issues*).

Next, to distinguish between the different complaint types, we measured the frequency of each complaint type. To do so, we follow the same approach used by Khalid et al. (2015), where we measure the percentage of reviews that belong to each complaint type on a per app basis. We calculate the percentage per app since different apps can have a different number of reviews, and if we do not normalize per app, then apps with more reviews could bias our results. Once we calculate the percentage of reviews for each complain type, we take the median percentage (from all the wearable apps) and assign it to the complaint type. Finally, we rank all of the complaint types from 1 - 15, where 1 is the highest (i.e., most frequent rank) and 15 is the least ranked.

The first three columns of Table 3.3 show the rank and median percentage of user reviews per complaint type. From the table we observe that complaints related to *Functional Errors* (i.e., bugs related to the functionality of the wearable app), *Cost* (i.e., issues related to the business model of the wearable app) and *Lack of Functionality* (i.e., deficiencies in the functionality of the app) are the most frequent complaints for wearable apps.

Our results highlight several new user complaint types that require attention from both, developers and software engineering researchers, such as: *Connection & Sync*, *Missing Notifications*, and *Device Compatibility* issues. It is important to acknowledge that wearable devices rely on handheld devices to perform expensive processing tasks, hence the connection and synchronization between them is critical. Our findings highlight areas where wearable apps need to address to ensure the high

Table 3.2: User Complaint Types and the Row Percentage of Reviews in each one.

Complaint Type	**Description**	**Example Review**	**%**
App Crashing	The wear app stops completely, goes idle or restarts	*"This app always crash on my phone."*	8.0
Battery Drainage	The wear app is draining the battery excessively	*"Worked less than half the time, and killed my Wear battery."*	7.4
Connection & Sync	Problems in connectivity with the wearable	*"Watch faces don't sync to watch. Uninstalling until this is fixed."*	18.1
Cost	Complaint about the wear app costs or business model	*"Have to purchase premium just to download anything."*	5.6
Device Compatibility	The wear app is not compatible with a given device	*"Won't work, only for famous smart watch."*	14.9
Feature Removal	A feature has been removed after an update	*"The latest update removed the watch battery state/graph. Why?"*	1.4
Feature Request	The user requires a specific new feature	*"Can we have a option to pick personal images for the top half of ..."*	3.0
Functional Error	A bug related to the functionality of the wear app	*"Dont buy...even the weather does not display correctly."*	26.1
Installation Problems	Issue while pushing the wear app to the wear device	*"App not pushed to watch."*	8.6
Lack of Functionality	Absence or deficiency of features in the wear app	*"Nothing special about this app and it's faces. They're barely acceptable..."*	11.4
Missing Notifications	The wear app lost or delayed notifications	*"Since I updated the app I get no notifications on either of my watches..."*	2.0
Performance Issue	The wear app slows or over use the resource	*"The app performs very poorly even after the 1.4 update."*	2.1
Privacy & Ethical	Invasion of privacy or ethical concerns complaint	*"Oh joy, more permissions and information gathering.. Smh"*	0.9
Spam Notifications	The wear app generates many unwanted nottifications	*"Keeps sending notifications to my watch telling me to download ..."*	2.6
UI Problems	Complaints about the interface design	*"Watch faces don't fit and are even off centre in compatibility mode."*	6.0
Uninformative	User reviews that do not have any useful information	*"It would not even let me play what even is this garbage???"*	8.2

Table 3.3: User Complaint Types Rank & Median Percentage for the Most Frequent and the Most Impactful Complaints.

Complaint Type	Most Frequent		Most Impactful	
	Rank	Median (%)	Rank	Median (1:2 star)
Functional Error	1	30.10	10	1.21
Cost	2	14.55	5	2.17
Lack of Functionality	3	14.22	8	1.46
Connection & Sync	4	10.03	4	2.63
Device Compatibility	5	9.57	2	4.10
UI Problems	6	7.34	14	0.78
Battery Drainage	7	7.06	12	1.06
App Crashing	8	6.38	6	2.06
Installation Problems	9	4.26	1	5.71
Feature Request	10	3.29	13	0.80
Spam Notifications	11	2.38	7	2.00
Performance Issues	12	1.98	15	0.65
Missing Notifications	13	1.65	11	1.17
Privacy & Ethical	14	1.12	3	3.17
Feature Removal	15	1.06	9	1.25

quality of apps. In particular, we recommend the development of tools and techniques that can assist developers with connection and sync issues and device compatibility issues. This seems particularly important for wearable apps, which typically require a mobile device for most useful features (e.g., sending out messages, or checking online resources). More generally, developers should be careful when pricing/advertising their apps, since cost-related complaints are frequent for wearable apps.

The most frequent complaints from the wearable app users are related to functionality errors, cost and lack of functionality.

3.3.2 Which User Complaint Types are the Most Negatively Impacting?

In addition to examining the frequency of the complaint types, we would also like to examine their potential negative impact. We examine the impact of each complaint type since, as previous work showed, the most frequent complaints may not be the most negatively impacting on the

users (Khalid et al., 2015). A negative impact can induce a snowball effect that will reduce the success of an app on its marketplace over time. For example, a complaint type that is very frequent, but that does not impact the users so much, may be better than a less frequent complaint type that has a large negative impact on the users. To study impact, once again, we follow the same methodology used by Khalid et al. (2015), where we measure the ratio of 1-to-2 star reviews for each complain type. Similar to the case when we calculate the frequency, we perform this calculation on a per app basis. Finally, we assign the median score from all the apps to the specific complaint type.

Table 3.3 (columns 4 and 5) show the of 1:2 star reviews for each complaint type. A 1:2 ratio of 1.21 shows that there are 21% more 1-star reviews assigned to this complaint type than 2-star reviews. A 1:2 ratio less than 1 indicates that there are more 2-star reviews assigned to the complaint type. Typically, higher ratio numbers can indicate a higher negative impact, and vice versa. From Table 3.3, we observe that the most impacting complaint types are the ones related to *Installation Problems*, achieving a 1:2 ratio of 5.71. In addition to *Installation Problems*, *Device Compatibility* issues, and *Privacy & Ethical Issues* also have a substantial negative impact on users. It is worth mentioning that compatibility to wearable devices is a challenge for developers to address, particularly since the app store does not provide a way to filter apps based on a specific wearable device; nor are developers able to provide multiple APKs based on the different wearable devices configurations (Android documentation, 2016b).

Our finding show that the most frequent user complaints are not necessarily the most impactful ones. A similar observation was made in the study by Khalid et al. (2015), in their study on mobile handheld device apps. For example, the *Installation Problems* complaint type has been ranked ninth in terms of number of complaints, while it is the highest impactful user complaint type.

Our finding that installation problems are highly impacting shows that wearable developers need to carefully test their apps, particularly the Android wear apps since, there are many Android devices that these wear apps need to be compatible with. Hence, we suggest the development of techniques that can address these compatibility issues, in particular issues that may impact the installation of wearable apps.

Table 3.4: Median and percentage values for Developer Replies and Reply Time in days per category.

Complaint Type	Developer Replies		Reply Time	
	Rank	Median (%)	Rank	Median (Days)
Privacy & Ethical	1	75.00	1	1
Performance Issues	2	58.33	14	4.5
Spam Notifications	3	50.00	6	2
Feature Removal	4	45.83	1	1
Missing Notifications	5	42.95	15	5.5
Cost	6	40.96	5	1.5
App Crashing	7	37.50	1	1
Device Compatibility	8	37.40	1	1
Installation Problems	9	37.30	6	2
UI Problems	10	36.93	13	4
Feature Request	11	33.33	12	3.5
Connection & Sync	12	31.77	6	2
Battery Drainage	13	25.00	11	3
Functional Error	14	22.22	10	2.5
Lack of Functionality	15	20.20	6	2

Installation problems, device compatibility issues and privacy & ethical issues are the most negatively impacting complaints.

3.3.3 What Types of Complaints Do Developers Reply to?

Thus far, our study has mainly focused on user complaints. However, the Google Play store provides the ability for developers to reply to user reviews in the hope of providing some clarification or support. Therefore, we mined a set of developer replies in order to get an answer for which types of user complaints developers care about the most. Complementing our user complaint data with their respective developer replies gives us a two-dimensional view of the issues that both, users and developers tend to care about.

Table 3.4 (columns 2 and 3) show the rank and median percentage value of developer replies for the different complaint types. From the table, we observe that *Privacy & Ethical Issues*, *Performance Issues* and *Spam Notifications* are the top three most replied-to complaint types. On the other

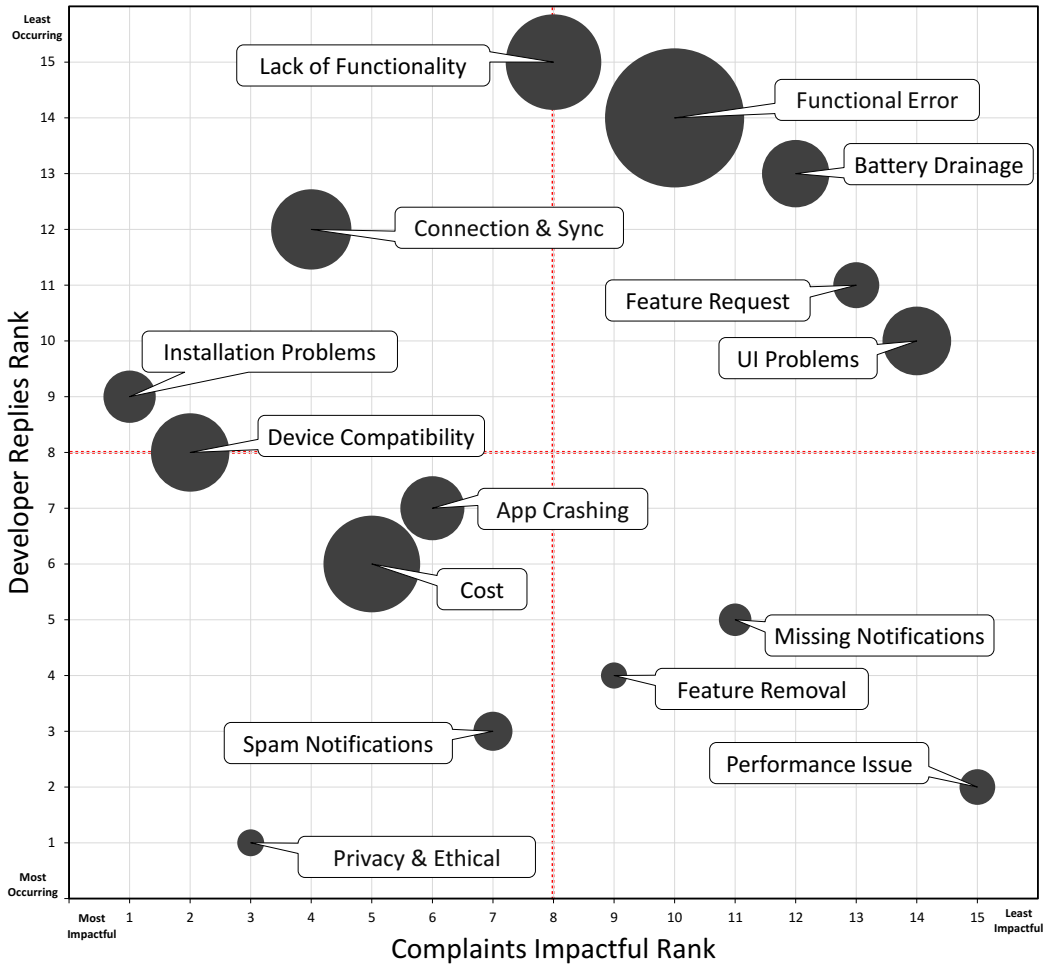


Figure 3.3: Impactful complaint types vs. developer replies.
The x-axis of the plot shows the rank of the most impactful complaint types. The y-axis shows the rank of developer replies, while the size of the bubble represents the frequency of each complaint.

hand, *Functional Errors* (which is the most frequent type of complaint) and *Installation Problems* (the most impacting) are not in the top most replied-to complaints. On the other hand, columns 4 and 5 in Table 3.4 show the rank and median time (in days) it took developers to reply to the different user complaint types. From the median reply time, we see that there are types of complaints that developers take longer to reply to, such as *Performance Issues*, *Missing Notifications* and *Feature Requests*.

In addition to the results presented in Table 3.4 and Table 3.3, we also use a Bubble plot to combine these three factors, i.e., complaint impact, frequency and developer replies; this is shown in Figure 3.3. The y-axis of the plot shows the rank in terms of developer replies, the x-axis shows the

rank in terms of impact of the complaint and the size of each bubble is used to represent frequency. The issues that have the most impact and receive the most replies are in the lower left quadrant, the issues that have an impact but do not get much developer attention are in the upper left corner, the issues that do not have a high impact but receive developer attention are in the lower right quadrant and finally, issues that do not have a high impact and do not receive much developer attention are shown in the upper right quadrant.

From Figure 3.3, we see that *Privacy & Ethical Issues*, *Spam Notifications*, *Cost* and *App Crashes* are important for both, users in terms of impact and receive replies from the developers. Complaints related to *Missing Notifications*, *Feature Removals* and *Performance Issues* tend to receive replies, however, they do not tend to have a significant impact on users. On the other hand, issues related to *Installation Problems*, *Device Compatibility*, *Connection & Sync Issues* and *Lack of Functionality* negatively impact users, but, developers do not tend to reply to them often. Lastly, complaints to *Functional Errors*, *Battery Drainage*, *Feature Requests* and *UI Problems* tend to be of low importance to both users (in terms of impact) and developers (in terms of replies).

One take away for developers from this research question is that they need to pay closer attention to complaints related to the aforementioned issues (e.g., *Installation Problems* and *Device Compatibility*), since those are generating the most negative impact on users. This is particularly important since previous studies showed that responding to user reviews affects the app's success (McIlroy et al., 2015; Palomba et al., 2015). From our results, we observe that developers should put additional effort in replying to these negatively 'impactful' complaint types to improve their app's ratings.

> ***The most frequent types of user complains that wearable app developers reply to are privacy & ethical, performance issues, and spam notifications. Furthermore, installation problems, device compatibility and connection & sync issues have higher impact but are not replied-to by developers.***

Table 3.5: Types of Developer Replies

Reply type	**Description**	**Example reply**
Request more details	The developer asking for more details	*"Can you record short video with this issue and send it to me [EMAIL]? Thank you."*
Notify that Issue is Solved	The issue already solved in a newer version	*"We just released v1.6.1 that fixes the problem, please update it and let us know if the problem goes away, thank you."*
Notify that a Solution is in Progress	Known issue and the developers work on it	*"I think I have identified the issue with crashes, should be fixed tomorrow."*
Provide a Solution / Workaround	Providing a solution to solve the issue	*"Hi ... , you can find steps for managing your notifications here: [WEBSITE] Let us know how it goes!"*
Offer Direct Support	The developer try to work directly on the case	*"Hello, please send me an email, I will help. Because of course it should work !"*
Offer Refund	The developer provide a refund offer	*"We will have a look at this. Let me know your order number to give you a refund."*
Other	General replies	*"Thank you for your feedback."*

3.3.4 How Do Developers Reply to Complaints?

In addition to quantifying the replies to the different complaint types, we also read through and classified the developer replies. In total, we had 782 replies. Similar to the case for the user reviews, we tagged each reply and added categories every time a reply did not fit into our existing categories. In the end, we ended up classifying the replies into 7 unique categories.

Table 3.5 shows the different reply categories, provides a brief description and an example of each reply category. From the table, we observe that most replies try to provide a solution or gather more information about the complaints. In the case of paid apps, developers may also offer a refund.

Figure 3.4 shows the percentage of replies in each category. The percentage is simply measured as the number of replies in a category over all the 782 replies. We find that the majority of the replies provide a solution to solve the user complaint, followed by replies that request more details about the issue mentioned in the review and replies to notify the user that a solution is in progress. Based on Figure 3.4, we see that the top four replies are related to the developers trying to get more information from the users, whereas, notification that a solution exists and offering a refund are the

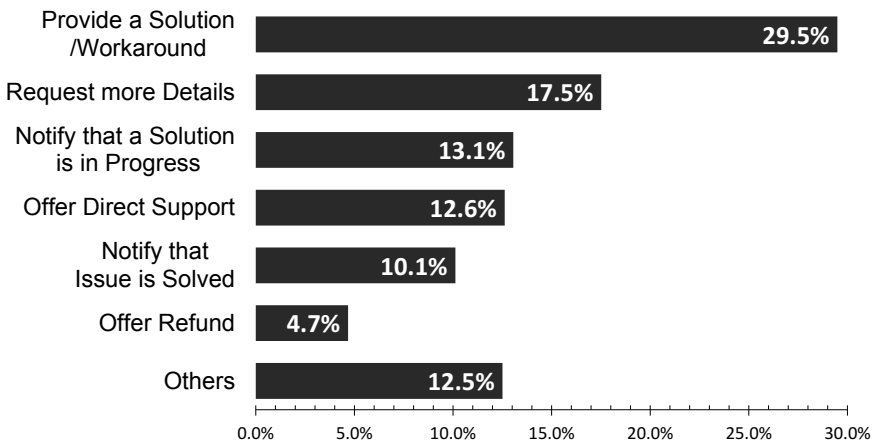


Figure 3.4: Percentage of developer reply types.

two least common replies.

Our results show that developers pay attention to the negative feedback given by users. When developers reply to low rated reviews, they do so to provide clarification or justification for missing features or problems in their wearable application. As previous studies have shown, developer replies tend to result in a positive update to the original low rating given by users (McIlroy et al., 2015). However, it is also important to consider that replying to the user is costly for developers. As we were able to observe from our dataset, the developer replies are manually generated; this problem needs to be addressed. A possible avenue for future work is to provide developers with a way to automatically respond to some of the most common complaints, which will lead to better reviews and minimal work for the developers.

> ***Wearable app developers mostly reply to user complains to provide a solution/-workaround, request more details and notify the user that a solution is in progress.***

3.4 Discussion

Thus far, we examined the user complaints for wearable apps. As mentioned earlier, prior work by Khalid et al. (2015) performed a similar study but for handheld devices. To determine the complaints that are specific to wearable apps and the complaints that are shared with handheld

Table 3.6: Comparison of Complaint Types for Wear and Handheld Devices. Based on the Findings Reported by Khalid et al. (2015)

Complaint Type	Frequency Rank		Impact Rank	
	Wearable	**Handheld**	**Wearable**	**Handheld**
Functional Error	1	1	10	7
Cost	**2**	**7**	**5**	**2**
***Lack of Functionality**	3	-	8	-
***Connection & Sync**	4	-	4	-
Device Compatibility	**5**	**8**	**2**	**5**
UI Problems	6	5	14	10
Battery Drainage	**7**	**12**	**12**	**8**
App Crashing	**8**	**3**	6	4
Installation Problems	9	-	1	-
Feature Request	**10**	**2**	13	12
***Spam Notifications**	11	-	7	-
Performance Issues	12	10	15	11
***Missing Notifications**	13	-	11	-
Privacy & Ethics	14	9	3	1
Feature Removal	**15**	**6**	**9**	**3**
**Netwrok Problem*	-	4	-	6
**Uninteresting Content*	-	11	-	9

devices, we now contrast our findings of complaint types for wearable apps to complaint types of handheld device apps.

3.4.1 Comparing Wear and Handheld Device User Complaints

Table 3.6 lists the complaint types found by our study and the ones mentioned in the study by (Khalid et al., 2015). Since the studies are done on a different sets of apps, we feel that it would be inappropriate to do the comparison of the percentages of reviews, therefore, we compare the complaint types in terms of their frequency and impact ranks. We observe from Table 3.6 that 11 of the 15 complaint types reported by our study are also mentioned in the study on handheld devices[3]. However, 4 complaint types appear only for wearable apps (marked in **bold** and with an * in the table) and 2 complaint types appear only for handheld devices (marked in *italics* and an * in the table).

[3]In some cases, the names of the complaints are different, however, we mapped them based on their descriptions, i.e., UI Problems is mapped to Interface Design, Battery Drainage is mapped to Resource Heavy and Performance Issues is mapped to Unresponsive App.

We see that complaints related to *Lack of Functionality* is mentioned, and with a high rank for wearable apps. The reason that it is mentioned for wearable apps and not handheld devices could be due to the fact most wearable devices are limited in what they can do, and heavily depend on the phone for any major features. This is perhaps disappointing for users, who in turn end up complaining about such issues. For example a user wrote: *"Useless app. Very limited features and designs"*. *Connection & Sync Issues* are also highly ranked and only mentioned for wearable apps. This problem was very clear from the reviews that we read. As mentioned earlier, in many cases the wearable apps heavily depend on the handheld devices and due to the fact that there exist a variety of wearable devices and a wide variety of handheld Android devices (Khalid et al., 2014), these *Connection & Sync Problems* are expected. For example a user stated *"No longer lets my Watch stay connected to my phone, completely ridiculous."* The problem is certainly magnified for wearable devices since they are limited to pretty much just displaying time and counting steps without a connection to the phone.

Other issues mostly related to notifications (i.e., *Spam Notification* and *Missing Notifications*) are also reported for wearable apps since this is one of the main ways that apps on the handheld device communicate with the user of the wearable app. As with any notifications, overdoing it causes users to complain. For example, *"Ads: No issues till it started popping up suggested watch faces in my notifications"*.

Moreover, we also highlight in bold the cases where the ranks for the same complaint types have a clear difference for wearable and handheld devices. Interestingly, we find that although *Cost* and *Battery Drainage* have a lower frequency rank for wearable apps, it has a higher impact rank compared to handheld devices (note that a lower rank score indicates higher importance). *Device Compatibility* is also different and has a lower frequency and impact rank for wearable apps. On the other hand, we find that *App Crashing*, *Feature Requests*, and *Feature Removals* have a higher frequency and impact rank for wearable apps. Further analysis as to why these differences exist is beyond the scope of this thesis, and warrants its own study. We argue that the reasons for the differences between both kinds of apps cannot be implied by looking at user reviews and their rating scores. This analysis requires to study additional factors such as hardware and API limitations that affect wearable and mobile development, which goes beyond the goals of our study.

Nevertheless, we do believe that our findings are the first to highlight such differences and open interesting questions regarding the differences between traditional handheld apps that run on phones and wearable apps that run on wearable devices.

3.4.2 Update-Related Complaints

A key observation presented in the paper by Khalid et al. (2015) is that many users posted complaints after an update. In handheld devices, update-related complaints account for approximately 11% of their studied reviews (Khalid et al., 2015). Similarly, we also noticed that many reviews mentioned problems after an update during our manual analysis. In fact, we found that approximately 12% of the examined wearable app complaints mentioned issues arising after an update. Our finding is similar to that reported by Khalid et al. (2015). A clear example of issues arising after an update are evident with the following review:

"Oh, my! My watch is completed useless again. Stop updating! Every time you fix a bug, you create many more!"

Although the study on handheld devices reported that most complaints after an update were related to functional errors, the addition/removal of a feature, and hidden costs, we found that most of the complaints for wearable apps were related to *Connection & Sync Issues* (32.9% of the reviews that report a problem after an update), *Functional Errors* (30.5%) and *Battery Drainage* (23%). For example, the user in the review below complains about connection problems since the last update of the app.

"Constantly drops connection to watch since update"

Our findings here draw attention to the importance of regression testing before an update is released. In particular, we suggest performing regression testing for *Connection & Sync* and *Battery Drainage Issues*.

3.5 Threats to Validity

Our study is subject to a number of internal threats and external threats to validity.

Internal Validity

To identify wearable app user complains, we manually classify 2,667 reviews. Like any human activity, the manual classification is susceptible to human error. To mitigate this threat, two researchers performed the manual classification. We also measured the agreement between the two annotators using Cohen's Kappa, which showed good agreement with value of +0.68.

Due to our manual classification phase being time consuming, we did not cover all of our data set, instead we took a sample of our dataset. This threat was addressed by taking a statistically representative sample with a 95% confidence level for each of the apps in our data set.

Our categorization is heavily dependent on the quality of the reviews provided by the users and their respective developer replies. As shown in prior studies, most user reviews contain useful information, however, in some cases different levels of details may lead to different complaint types.

Martin, Harman, Jia, Sarro, and Zhang (2015) studied a common problem of sampling bias when research work analyzes data mined from app stores. This problem exists because of an often-limited access to a full set of apps and their reviews to be studied. When studies are done only on subsets of data, they can be potentially biased and draw non-reasonable conclusions.

In the scope of our work, despite our efforts in the data crawling phase, although we did not face the limitations for crawling described by Martin et al. (2015) in the Google App Store, we do not claim to have a guaranteed full set of reviews for each app. Furthermore, our work does target user complains only, which is already a given subset of user reviews; this is unavoidable for our purpose. However, to address the threat of bias, we took statistically significant random samples of the reviews for each app we study. These measures were taken precisely to remove bias from the study while following the similar approach previously used by Khalid et al. (2015)

It is important to note that throughout our study, we use the low ratings given by user reviews, i.e., 1 or 2 stars ratings, as a way to assess impact. We used this definition of impact, since it was used by Khalid et al. In their work. That said, we do believe that other definitions for impact are possible. For example, the messages of the reviews could be analyzed to determine the sentiment expressed by users; this can be done using a tool such as *Sentistrength* (Thelwall, Buckley, Paltoglou, Cai, & Kappas, 2010). In the future, we plan to explore other ways of measuring impact of a review.

External Validity

We found over 17,000 wearable app related user reviews but we filtered them down to 5,751, and hence, our data set can be considered small. This however, is because this platform is fairly new and we were only able to select the 19 wearable apps that had over 100 user reviews to make our findings from them relevant. On the same line of thought, the filtering phase for the wearable app related reviews may have discarded some useful information that did not match our filtering rules. Moreover, our study is performed on Android Wear apps, hence our findings may not generalize to wearable apps from other platforms.

3.6 Conclusion

Users provide direct feedback on their experience of mobile apps through user reviews. Prior work showed that user reviews can be mined to effectively determine user complaints to help developers understand the issues that users of handheld apps face the most, so they can be mitigated.

Given that wearable apps are a new trend that is only increasing in popularity, in this chapter, we mine user reviews in order to understand the user complaints of wearable apps. We manually sample and categorize 2,667 reviews from 19 wearable apps. We find 15 unique complaint types that wearable users report in user reviews. We also examine the replies that developers post to some of the user complaints in order to determine complaints that developer care most about and identify areas that are important for users, but are not well replied-to by developers.

Our findings indicate that the most frequent complaints are related to *Functional Errors*, *Cost* and *Lack of Functionality*. On the other hand, we find that developers reply most to complaints related to *Privacy & Ethical Issues*, *Performance Issues* and notification-related issues. And, when developers reply they mostly do so to provide a solution, request more details or let the user know that they are working on solving the problem. We also compare our findings on wearable apps with the study by Khalid et al. (2015) on handheld devices and find that 1) that 11 of our 15 complaint categories are also reported for handheld devices and 2) similar to the case of handheld devices, approximately 12% of complaints mention an update.

In this chapter our findings show that the most negatively impacting complaints are related to

Installation Problems and *Device Compatibility*. After some investigation, we find that the permission model used in Android contributes to these negatively impacting issues. Hence, in the next chapter, we perform an empirical study to examine these permission related issues in wearable apps, particularly issues related to permission mismatches and superfluous features. In addition, we built a tool that implements our technique to automatically detect these permission problems from an app's APK file, which is used to perform our empirical study.

Chapter 4

Detecting Permission Problems of Wearable Apps

4.1 Introduction

Mobile apps are playing an increasingly important role in our daily lives. These mobile apps can monitor all kinds of activities, e.g., get our location, the interaction with contacts, etc. More recently, wearable devices that run wearable apps have enhanced the capabilities of these mobile apps. However, due to their 'always connected' nature, privacy and security issues have become a very important issue.

To help protect users' privacy and make sure that apps do not intentionally or unintentionally access data that does not need to be shared, permissions are used to control what an app can access. For example, if an app needs access to the camera or microphone, it needs to explicitly ask for this permission in its configuration files (i.e., AndroidManifest.xml file).

Therefore, a plethora of prior work studies permission management in mobile apps. For example, work by Au et al. (2012), Stevens et al. (2013) and Jha et al. (2017) 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 (Book, Pridgen, & Wallach, 2013; Dering & McDaniel, 2014; Enck, Ongtang, & McDaniel, 2009; Watanabe, Akiyama,

Sakai, & Mori, 2015), and discover permission misuses (Au et al., 2012; Felt et al., 2011; Stevens et al., 2013), studying the evolution of permissions over time (Calciati & Gorla, 2017; X. Wei et al., 2012), and suggesting which permission should be requested (Bao et al., 2016; Bao, Lo, Xia, & Li, 2017; Karim et al., 2016). All of the aforementioned work focused on the permissions of mobile apps exclusively.

To make a bad situation even worse, the introduction of wearable devices and apps further complicates the permission management since both, 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[1]). If fact, our findings in the previous chapter showed, that indeed installation and device compatibility are some of the two most impacting issues. However, to the best of our knowledge, no work has examined the permission issues related to wearables apps.

In this chapter, we conduct an empirical study to examine the permission problems in the context of wearable apps. In particular, we focus on two of the most common issues: First, **permission mismatch**, which refers 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 the installation step or throw a security exception. Second, **superfluous features**, which refers to the case where an app requests a permission that requires a specific hardware resource (e.g., access to the microphone) without really needing 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.

We perform our study on 2,724 apps from the Google Play store that contain a wearable version. Our findings show that the permission mismatches exist in 6.1% of the released apps on the app store. Moreover, we find that 19.2% of studied wearable apps contain superfluous features. To operationalize our work we built a tool, called PERMLYZER, that automatically detects these two problems from Android APKs. PERMLYZER can be leveraged by developers to ensure that their APKs do not suffer from any permission related issues prior to release.

This chapter makes the following contributions:

- To the best of our knowledge, this the first study to examine permission problems in the

[1]For versions before Android 6.0 (API level 23)

context of wearable apps.

- We define and examine the two main permission problems - namely the permission mismatch problem and superfluous features; then we perform an empirical study to examine them by investigating 2,724 embedded and 339 standalone wearable apps.

- We implement our approach in a tool called PERMLYZER, which will be freely available; also, we provide our dataset of the crawled and analyzed apps publicly available.

The remainder of this chapter is organized as follows: Section 4.2 provides a background about Android platform and wearable related concepts. Section 4.3 describes the problems that related to wearable apps permission model, and Section 4.4 illustrates our approach. In Section 4.5, we show the findings of our empirical study, and discuss some in Section 4.6. We discuss the limitations of our study in Section 4.7. Lastly we conclude the chapter and sketch future work in Section 4.8.

4.2 Background

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

4.2.1 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. Android apps are distributed mainly through the Google Play store as APK archive files. Every APK must contain 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 describes the app components, 2) it requests the permissions, 3) it declares the required hardware or software features, and 4) it specifies the minimum and target API level (Android documentation, 2017a).

Wearable apps on the Android platform can be distributed in two ways: 1) by embedding the wearable app inside a handheld app; or 2) publishing more than one APK under the same app

listing using the multiple APK feature of Google Play store, so when a user installs the handheld app the platform will automatically install the wearable app on the paired wearable device (Android documentation, 2017e, 2017i). The Android platform also provides a framework of application programming interfaces (APIs) that apps can use to interact with the underlying functionality of the platform. 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 change to higher number. The new API remains compatible with all earlier API Levels. Therefor, apps that design for an API level can run on a higher level, but it cannot run on lower level (Android documentation, 2017k).

4.2.2 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 user experience (Android documentation, 2017a). 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 4.1 is an example of requesting 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 should follow when determining whether or not to grant the permission to an application requesting it. The Android platform automatically grants permissions from the *Normal* protection level (i.,g. `BLUETOOTH`) at installation, without asking for the user's explicit approval; and *Dangerous* permissions (i.,g. `CAMERA` and `MICROPHONE`) that are requested by an app might be displayed to the user and require confirmation before it is being granted. Also, third-party apps can ask for permissions from both *Normal* and *Dangerous* protection level categories.

Listing 4.1: Example illustrate the AndroidManifest.xml file

```
<manifest . . . >
  <uses-permission android:name="android.permission.READ_SMS" />
  <uses-feature android:name="android.hardware.telephony"
      android:required="true" />
       . . .
  <application . . .>
    <activity android:name="com.example.project.FreneticActivity"
        android:permission="com.example.project.DEBIT_ACCT" . . . >
        . . .
    </activity>
  </application>
</manifest>
```

Specific to wearable apps, developers have to match permissions that are requested in the wearable version with permissions requested in handheld version. In other words, all requested wearable permissions have to also be listed in the manifest file of the handheld app (Android documentation, 2017e). However, the release of Android 6.0 (API level 23) has 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 (Android documentation, 2017g, 2017h). These changes affect how wearable app permissions are declared.

4.2.3 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, platform version and screen configuration. To reach the largest possible user-base for an app, developers attempt to support as many device configurations as possible. Unfortunately supporting all device configuration is sometimes not possible. 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 Google Play store based on the device characteristics (Android documentation, 2017c, 2017d). 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 (Android documentation, 2017c). 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 file `AndroidManifest.xml`. The developers declare the features that their app 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 (Android documentation, 2017j). Line 3 in Listing 4.1 is an example of a feature deceleration 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 a `<uses-feature>` element for `android.hardware.microphone`, Google Play store considers that the application requires a microphone and should not be shown to users whose devices do not have a microphone (Android documentation, 2017j). To avoid setting filters for hardware features that the app can operate without them, the app developer could declare the underlying features in the manifest file and give the value `false` to the attribute `android:required`.

4.3 Problems

To illustrate the challenge of dealing with permission problems in the context of the wearable app development. We first enhance the discussion by presenting an example. Second we present the main two wearable apps permission problems addressed in this chapter.

Motivation 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 of sending 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 inherits the permissions that the platform granted to the handheld app.

Failing to match the permission could cause problems since the wearable app cannot get the required permissions. On the other hand, considering the handheld app request the permission `SEND_SMS`, the Google Play store will consider it as depending on a telephony functionality even if the app that do not request the feature `android.hardware.telephony`. Hence, the handheld app will not be available in devices that does not have the telephony functionality such as most of tablet devices.

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

4.3.1 Permission Mismatches

Description. To distribute a wearable app to users, the wearable APK could be embedded in a handheld APK. And 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 at 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 should

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 (Android documentation, 2017f). 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 version of the app; we call this the ***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 (Mujahid, 2017). Additionally, the permission mismatch problem is particularly problematic since: 1) it does not raise compilation errors or log any message in the `logcat`; 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 most 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. Subsequently, it leads to a negative user experience, which is reflected in low ratings and revenues.

4.3.2 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 (Android documentation, 2017j).

Features and permissions that an app declares in the manifest file could 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 that require underlying features even if it not declared. Thus, requesting a permission in the manifest file could cause Google Play store to set

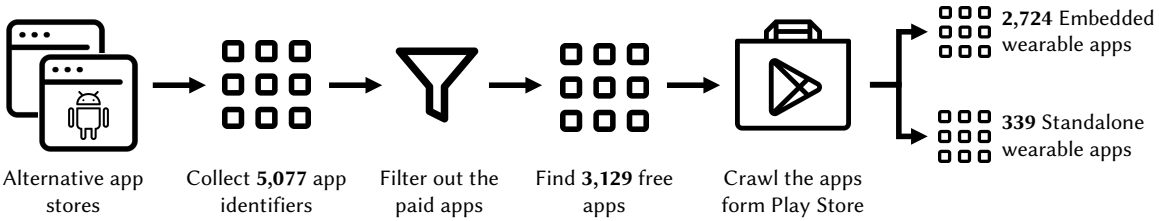


Figure 4.1: The overview of data collection process.

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 should 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.

Implcation. 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 them. Moreover, the underlying features could belong to unused permissions which the app requests them without using the functionalities that they grant to the app, we call such a case the ***superfluous feature***. As result, the superfluous features set unexpectedly filters out legitimate compatible devices, which reduces the potential customer base for the app, negatively impacting its revenues.

4.4 Study Setup

In this section, we detail our dataset collection, and then we describe the approach as it illustrated in Figure 4.2. Each step of the approach is detailed in the the following subsections.

4.4.1 Dataset

As it is shown in Figure 4.1, we select the available Android Wear apps on Google Play store by collecting their identifiers from two alternative app markets: *Android Wear Center* (Wearable Software, 2016) and *GoKo Store* (Korner et al., 2016) that accessed on September 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 using our technique. In order

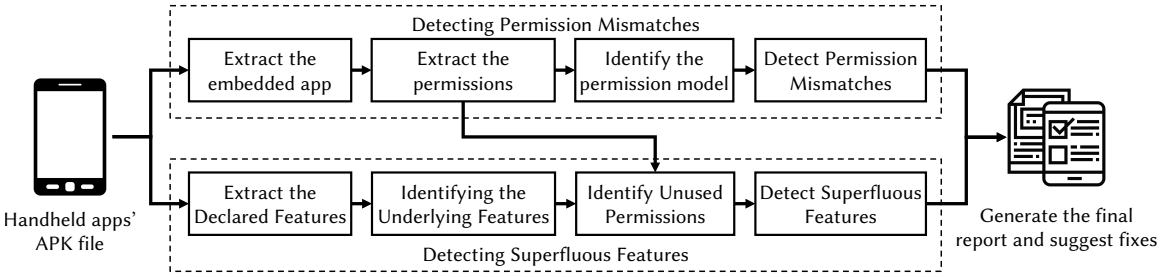


Figure 4.2: The approach overview.

to download the last version of the selected apps, we built a crawler that interfaces with the Google Play store API as a regular mobile device to download the handheld apps and as a wearable device to download the 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. After downloading and unpacking the apps, we find 2,724 apps contains an embedded wearable app and 339 apps have a standalone wearable app.

4.4.2 Detecting Permission Mismatches

Devices that run API level lower than 23 require wearable apps to match their permissions with the permissions request in handheld version. Nevertheless, in some cases, the wearable app developers may request permissions that do not exist in the handheld app, resulting a permission mismatch.

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 of them. 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 examine the permission of the wearable and the handheld app. The following subsections are the detailed steps to analyze APK files of an wearable app and detect the permission mismatch problem. Figure 4.2 illustrates the approach overview.

Extract the Embedded Wearable APK

A standard distribution model for Android Wear is embedding a wearable app inside a handheld app. When users install the handheld app, the Android platform pushes the wearable app to the paired watch (Android documentation, 2017e).

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 (Tumbleson & Winiewski, 2017); a tool for reverse engineering Android apps. The tool allows to retrieve nearly 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 meta data tag that refers to the wearable app description file[2] 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 several 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] A file that contains the version and path information of the wearable app APK.

Extract the Requested Permissions

First, we parse the file `AndroidManifest.xml` from both of the handheld app and the embedded wearable app. Second, for both manifests we select the permissions through targeting the tags `<uses-permission>`; then for each tag we read the value of the attribute `android:name`. Finally, we check if the permission are belong to Android Open Source Project (AOSP) or to a third party app.

Identify the Permission Model

Since not all wearable apps are required to match their permission, we check which permission model that the app should implement. To do so, we select the tag `<uses-sdk>` from manifest file of the handheld app and we read the value of its attribute `android:minSdkVersion`. If the value is lower than 23, then the app should match the permissions of its wearable version with the handheld version's permissions.

Detect 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 a permission mismatch.

4.4.3 Detecting Superfluous Features

Some permissions grant a functionality that depends on hardware features. To prevent apps to be installed on incompatible devices, the Google Play store assumes that the app requires the underlying features of the requested permissions. So, such apps will not be available on devices that do not have/support that hardware features associated with the requested permissions. However, apps may request permissions that they do not need in the first place, resulting in what we call, 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 4.2 illustrates an overview of our approach.

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 4.4.2. 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.

Identifying the Underlying Features

We describe in Section 4.3.2 that some permissions depend on hardware features, e.g., if an app requests the permission `SEND_SMS`, it means the app depends on `Telephony` API (Android Developers Reference, 2017). Thus, the app should declare the feature `android.hardware.telephony` in its manifest file; if not, we consider such case as ***missed underlying feature***.

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 called APKANALYZER (Android Studio, 2017a) from the Android's Software Development Kit (SDK). Among other things, this tool extracts features that trigger the filtering of 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.

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 that 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 (Au et al., 2012), 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 (Desnos & Gueguen, 2017) 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 4.4.2 ; 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***.

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.

4.4.4 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-generate a new `AndroidManifest.xml` file that implements the suggested fixes.

4.5 Results

The main goal of our study is to detect permission problems in the context of wearable apps. Although some prior work examined the permission problems in mobile apps (Martin et al., 2017), to the best of our knowledge, this is one of the first studies to exclusively focus on the permission problems in the context of wearable apps. Furthermore, in addition to defining the main problems related to permission in wearable apps, we also examine and quantify these permission problems in the apps published on 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

Wearable apps need permissions to access sensitive data and functionalities, e.g., read contacts or access heartbeat sensor. While the wearable app is embedded in a handheld app, the user grants the permissions to the handheld app and the wearable app inherits them. For this reason developers match the wearable permissions with the handheld permissions. In production, failing to do so causes a permission mismatch problem, which affects the ability of running the wearable apps properly on wearable devices. Hence, our goal is to quantify the amount of apps that suffer from this permission mismatch problem.

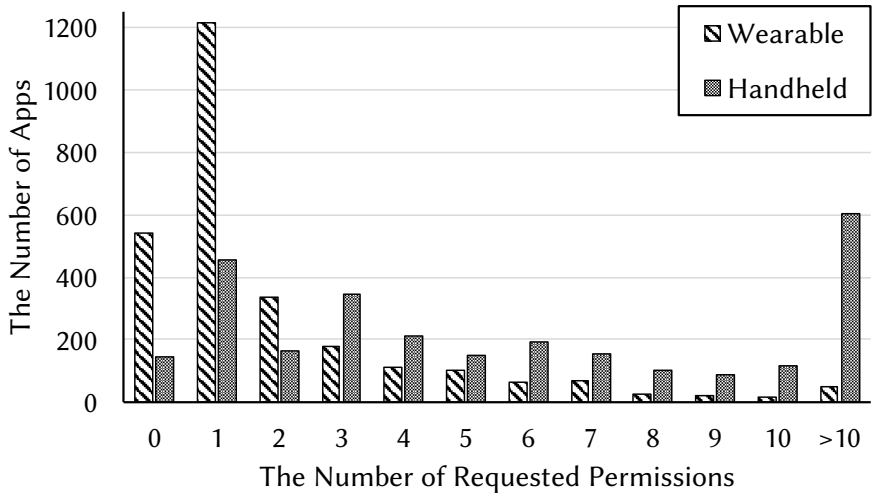


Figure 4.3: Histogram showing the number of wearable and handheld apps at each level of requested permissions.

Approach

To answer this research question, we use the tool PERMLYZER that internally applies the approach that we described in Section 4.4.2 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 4.3 shows the amount 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.

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

Table 4.1: 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

READ_EXTERNAL_STORAGE in the wearable app, but does not request the same permission in the handheld version.

We also investigate what type of permissions are most likely to be mismatched. Table 4.1 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.***

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

Motivation

Running an app on incompatible devices can negatively impact the user experience. To avoid such cases, developers should always declare all features that their apps are using or requiring (Android documentation, 2017j). Apps could use APIs that depend on hardware features of the device. In order to have access to those APIs, the app should request the suitable permissions in its manifest file. We call these hardware features that the permission depends on, the underlying feature. To

avoid unintentionally making the app available on incompatible devices, the Google Play store considers the underlying features of requested permissions as required by default (Android documentation, 2017j). Some apps can operate or are designed to function even if part/all of the underlying features are not available. So, missing the declaration of underlying features could be problematic since the default behavior of the Google Play store could limit the number of devices that can access the app. Thus, it is important for wearable apps to declare the underlying features of all requested permissions to avoid having superfluous features.

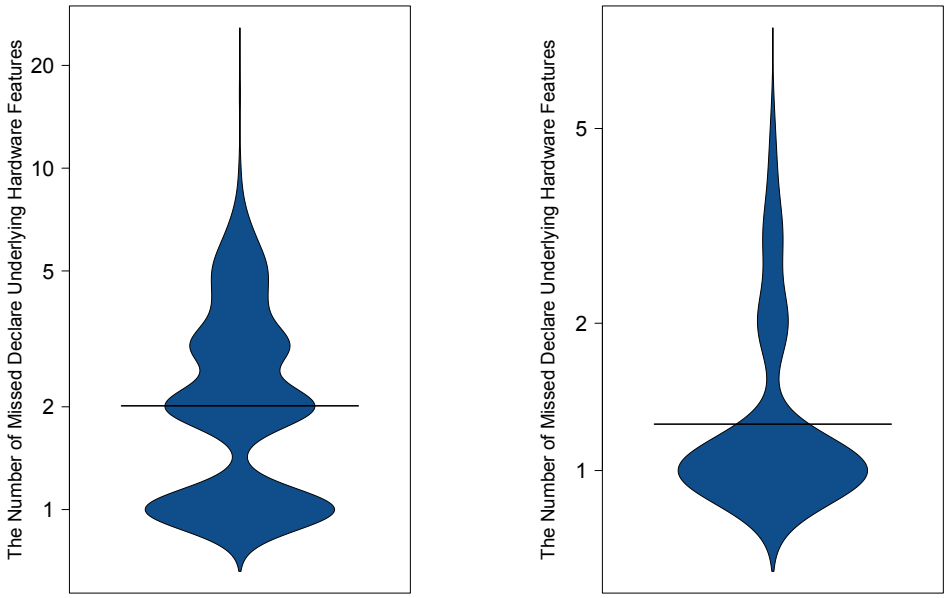
In this research question, first we want to study how apps declare their hardware features for their requested permissions. Thus, we examine the use of this functionality in the published wearable apps on the Google Play store. Second, we study the missed underlying features and examine if they are superfluous 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 deceleration 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 most of the apps declare some of their hardware features, 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 4.4 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, in median they



(a) Handheld (b) Standalone wearable

Figure 4.4: The number of missed underlying features in the studied apps.

missed 1 hardware feature and at max 5 features.

To emphasize the underlying feature declarations, Table 4.2 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 deceleration 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 the declare its underlying feature `android.hardware.wifi`. As a result, the app is not available for the wearable devices that do not support the Wi-Fi connectivity.

Also, we find that 523 (52.4%) handheld apps and 66 (80.5%) standalone wearable apps have at least one superfluous feature. For example, the app `com.runar.wearcompass` 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 4.3 shows the list of permissions that introduce the superfluous feature in the studied

Table 4.2: The underlying features with the percentage of handheld and standalone wearable apps that declared/missed the underlying features.

	Handheld (%)		Standalone (%)	
Feature Name	**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	-

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 4.4 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.

4.6 Discussion

In this section, we discuss the problem of unused permissions in the wearable apps and its negative impacts. And also we shed light on the differences in the notion of the wearable apps that the research community should consider when they study the permissions of the Android apps.

Table 4.3: 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

Table 4.4: 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

4.6.1 Does matching the permissions contribute to introduction of 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) (Calciati & Gorla, 2017; X. Wei et al., 2012). As we describe in the Section 4.2.2, 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 do 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. (2011) studied 795 mobile apps and showed that 32.7% of them have unused permissions. More recently, X. Wei et al. (2012) 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 the comparison shows, the permission matching requirement can be a factor to introduce unused permissions.

4.6.2 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 (X. Wei et al., 2012).

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 (Book et al., 2013; Dering & McDaniel, 2014; Enck et al., 2009; Watanabe et al., 2015), evolution over time (Calciati & Gorla, 2017; X. Wei et al., 2012), discover permission misuses and overprivilegesd (Au et al., 2012; Felt et al., 2011; Stevens et al., 2013) and suggest which permissions should be requested (Bao et al., 2016, 2017; Karim et al., 2016). To best of our knowledge, previous studies do not consider the notion of wearable apps when they perform analysis that deal 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 that 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.

4.7 Threats to Validity

In this section, we disuses the 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 (Desnos & Gueguen, 2017) tool to extract the platform API calls using a static analysis approach and on the mapping of PSCOUT (Au et al., 2012) tool to link the platform APIs with the corresponding permission. To help alleviate this threat, we manually investigated some of the result reported unused permissions and in all cases the manually examined cases were correct.

Our results also include only AOSP permissions in the process of analyzing unused permissions. Other third party permission could have different pattern in term of declare the unused permissions.

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 does not reflect the problems in apps that set filtrates that prevent the them to be available for the device that our script used it face Google Play store in the crawling process.

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%[3] of devices running on Android version with API level lower than 23 (Android documentation, 2017b), which highlight the importance of supporting such devices.

4.8 Conclusion

Wearable devices' popularity is increasing. In fact, based on our data collection, the Google Play store contains more than 5,077 wearable apps. In this chapter we defined two permission problems related to wearable apps - namely permission mismatches and superfluous features. To mitigate the problems, we propose a technique to detect permission problems in wearable apps. We implement

[3]Data collected during a 7-day period ending on October 2, 2017

our technique in a tool called PERMLYZER, which automatically detect these permission problems form an app's APK. We run PERMLYZER on top of 2,724 app 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.

Chapter 5

Summary, Contributions and Future Work

This chapter concludes the thesis. We presents summary of results presented throughout this thesis. We then discuss possible directions for future work.

5.1 Summary of Addressed Topics

This thesis focuses on the challenges of software development of wearable apps. First, we conduct an empirical study to understanding the main issues in developing wearable apps and their impact on the apps' users. Then, we propose a technique to detect permission problems, which raise issues that have the most negative impact on wearable app users and built a tool that automatically detects these problems, called PERMLYZER.

The following are the summaries of this thesis chapters.

Chapter 3 presents what wearable apps' users complain about. In this chapter, we study users' complains through the examination of users reviews posted on Google Play store. We conducted an qualitative analysis where we analyze 2,667 user's review. We found that: 1) *Functional Errors*, *Cost*, and *Lack of Functionality* are the three most frequent complaints; 2) *Installation Problems*, *Device Compatibility*, and *Privacy & Ethical Issues* are the most negatively perceived by users; 3) developers are most likely to reply to complaints related to *Privacy & Ethical Issues*, *Performance*

Issues, and *Spam Notifications*. We also contrast the complaints based on their impact and the developer replies and find that *Installation Problems*, *Device Compatibility*, and *Connection & Sync Issues* are most impacting, but have a low response rate from developers.

Chapter 4 presents the main permission problems in wearable apps. In this chapter, we studied the problem of permissions in the context of wearable apps. We identified the main permission problems, and we conducted an empirical study to examine the prevalence of these problems and their impact on real-world wearable apps. Our findings show that: 1) 6.1% (132) of wearable apps that request permissions are suffering from the permission mismatching problem, which cause falling the app to be installed on the wearable device or throws a security exception; 2) 19.2% of the studied wearable apps contain superfluous features, which causes the Google Play store to filter out devices that do not support/have this hardware feature, reducing the potential customer base for the app.

5.2 Contributions

The major contributions of this thesis are as follows:

- Empirically investigate user complaints in wearable apps by manually classify 2,667 reviews belonging to 19 wearable apps; then measured the frequency of the complaints and how negatively they are perceived by users.
- Examine the developer replies to the studied complaints in order to better understand the areas that receive enough attention and areas that are important to the users.
- Define and examine the two main permission problems that are related to the most negatively impactful user complaints; then perform an empirical study to examine them through investigating 2,724 embedded apps and 339 standalone wearable apps.
- Implement our approach of detecting wearable permissions' problems in a tool called PERMLYZER, which will be freely available.
- Provide our dataset of the crawled apps' APKs, the detailed analyses data, the collected 1.2 million reviews, and the manually classifying of the reviews to be publicly available.

5.3 Future Work

We believe that our thesis makes a positive contribution towards the goal of understanding the challenges of wearable apps development. However, there are still many open challenges that need to be tackled to improve the development of wearable apps. We now highlight some avenues for future work.

5.3.1 Considering other aspects to assess the impact of user complaints

Throughout our study, we use the low ratings given by user reviews, i.e., 1 or 2 stars ratings, as a way to assess impact. We do believe that other definitions for impact are possible. For example, the messages of the reviews could be analyzed to determine the sentiment expressed by users. In the future, we plan to explore other ways of measuring impact of a user review.

5.3.2 Expanding the scope of our tool

In our investigation in this thesis, we observe many other problems related to the spatiality of wearable apps; e.g., improper configuration of the type of wearable app, registration of unused broadcast receiver, or definition of deprecated wearable filters. Such problems affect the availability, performance and quality of wearable apps. Addressing these problems can positively impact the the wearable apps development.

5.3.3 Measuring the impact of the requested permissions and features

Also, it will be helpful for a developers to receive real-time feedback about the impact of the each feature or permission that they request, showing the number of potential users' devices that will be eliminated for his/her app. Using the same technique, an empirical study can help to understand the evolution of permissions and features across the apps' versions over time.

5.3.4 Extending to other platforms

Our study is performed on Android Wear apps, hence our findings may not generalize to wearable apps from other platforms. So, a future work will be to examine other platforms and explore

the differences in term of problems and complaints.

References

Ahola, J. (2015). Challenges in android wear application development. In *Proceedings of the 15th international conference on web engineering* (pp. 601–604). Springer.

Android Developers Reference. (2017). *Telephony.* https://developer.android.com/reference/android/provider/Telephony.html. (Accessed on December 1, 2017)

Android documentation. (2016a). *Creating wearable apps.* https://developer.android.com/training/wearables/apps/index.html. (Accessed on October 2, 2016)

Android documentation. (2016b). *Filters on google play.* https://developer.android.com/google/play/filters.html. (Accessed on December 18, 2016)

Android documentation. (2017a). *App manifest.* https://developer.android.com/guide/topics/manifest/manifest-intro.html. (Accessed on November 30, 2017)

Android documentation. (2017b). *Dashboards.* https://developer.android.com/about/dashboards/index.html. (Accessed on October 31, 2017)

Android documentation. (2017c). *Device compatibility.* https://developer.android.com/guide/practices/compatibility.html. (Accessed on October 4, 2017)

Android documentation. (2017d). *Multiple apk support.* https://developer.android.com/google/play/publishing/multiple-apks.html. (Accessed on October 4, 2017)

Android documentation. (2017e). *Packaging wearable apps.* https://developer.android.com/training/wearables/apps/packaging.html. (Accessed on January 19,

2017)

Android documentation. (2017f). *Packaging wearable apps.* https://developer.android.com/training/wearables/apps/packaging.html. (Accessed on January 19, 2017)

Android documentation. (2017g). *Requesting permissions at run time.* https://developer.android.com/training/permissions/requesting.html. (Accessed on October 4, 2017)

Android documentation. (2017h). *Requesting permissions on android wear.* https://developer.android.com/training/articles/wear-permissions.html. (Accessed on October 31, 2017)

Android documentation. (2017i). *Standalone apps.* https://developer.android.com/training/wearables/apps/standalone-apps.html. (Accessed on November 30, 2017)

Android documentation. (2017j). *uses-feature.* https://developer.android.com/guide/topics/manifest/uses-feature-element.html. (Accessed on October 6, 2017)

Android documentation. (2017k). *uses-sdk element.* https://developer.android.com/guide/topics/manifest/uses-sdk-element.html. (Accessed on October 4, 2017)

Android Studio. (2017a). *Apk analyzer tool.* https://developer.android.com/studio/command-line/apkanalyzer.html. (Accessed on December 1, 2017)

Android Studio. (2017b). *Improve your code with lint.* https://developer.android.com/studio/write/lint.html. (Accessed on June 11, 2017)

Au, K. W. Y., Zhou, Y. F., Huang, Z., & Lie, D. (2012). Pscout: Analyzing the android permission specification. In *Proceedings of the acm conference on computer and communications security* (pp. 217–228). ACM.

Bao, L., Lo, D., Xia, X., & Li, S. (2016, Nov). What permissions should this android app request? In *Proceedings of international conference on software analysis, testing and evolution* (p. 36-41).

Bao, L., Lo, D., Xia, X., & Li, S. (2017, Jul 28). Automated android application permission recommendation. *Science China Information Sciences*, *60*(9), 092110.

Barrera, D., Kayacik, H. G., van Oorschot, P. C., & Somayaji, A. (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* (pp. 73–84). ACM.

Bonato, P. (2010, May). Wearable sensors and systems. *IEEE Engineering in Medicine and Biology Magazine*, *29*(3), 25-36.

Book, T., Pridgen, A., & Wallach, D. S. (2013). Longitudinal analysis of android ad library permissions. *arXiv preprint arXiv:1303.0857*.

Calciati, P., & Gorla, A. (2017, May). How do apps evolve in their permission requests? a preliminary study. In *Proceedings of 14th ieee/acm international conference on mining software repositories* (p. 37-41).

Chauhan, J., Seneviratne, S., Kaafar, M. A., Mahanti, A., & Seneviratne, A. (2016). Characterization of early smartwatch apps. In *Proceedings of the 2016 ieee international conference on pervasive computing and communication workshops* (pp. 1–6). IEEE.

Chen, N., Lin, J., Hoi, S. C., Xiao, X., & Zhang, B. (2014). Ar-miner: mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th international conference on software engineering* (pp. 767–778). ACM.

Ciurumelea, A., Schaufelbhl, A., Panichella, S., & Gall, H. C. (2017, Feb). Analyzing reviews and code of mobile apps for better release planning. In *Proceedings of the 24th ieee international conference on software analysis, evolution and reengineering* (p. 91-102). IEEE.

Cohen, J. (1960). A coefficient of agreement for nominal scale. *Educational and Psychological Measurement*, *20*, 37–46.

Dering, M. L., & McDaniel, P. (2014). Android market reconstruction and analysis. In *Proceedings of the ieee military communications conference* (pp. 300–305). IEEE Computer Society.

Desnos, A., & Gueguen, G. (2017). *Androguard: Reverse engineering, malware and goodware analysis of android applications).* https://github.com/androguard/androguard. (Accessed on November 27, 2017)

Di Sorbo, A., Panichella, S., Alexandru, C. V., Shimagaki, J., Visaggio, C. A., Canfora, G., & Gall, H. C. (2016). What would users change in my app? summarizing app reviews for recommending software changes. In *Proceedings of the 24th acm sigsoft international symposium on foundations of software engineering* (pp. 499–510). ACM.

Di Sorbo, A., Panichella, S., Alexandru, C. V., Visaggio, C. A., & Canfora, G. (2017). Surf: Summarizer of user reviews feedback. In *Proceedings of the 39th international conference on software engineering companion* (pp. 55–58). IEEE Press.

Do, Q., Martini, B., & Choo, K.-K. R. (2017). Is the data on your wearable device secure? an android wear smartwatch case study. *Software: Practice and Experience*, *47*(3), 391–403.

Enck, W., Ongtang, M., & McDaniel, P. (2009). On lightweight mobile phone application certification. In *Proceedings of the 16th acm conference on computer and communications security* (pp. 235–245). ACM.

Felt, A. P., Chin, E., Hanna, S., Song, D., & Wagner, D. (2011). Android permissions demystified. In *Proceedings of the 18th acm conference on computer and communications security* (pp. 627–638). ACM.

Finkelstein, A., Harman, M., Jia, Y., Martin, W., Sarro, F., & Zhang, Y. (2017). Investigating the relationship between price, rating, and popularity in the blackberry world app store. *Information and Software Technology*, *87*, 119–139.

Fleiss, J. L., & Cohen, J. (1973). The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and Psychological Measurement*, *33*, 613–619.

Fu, B., Lin, J., Li, L., Faloutsos, C., Hong, J., & Sadeh, N. (2013). Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th acm sigkdd international conference on knowledge discovery and data mining* (pp. 1276–1284). ACM.

Galvis Carreño, L. V., & Winbladh, K. (2013). Analysis of user comments: An approach for software requirements evolution. In *Proceedings of the 2013 international conference on software engineering* (pp. 582–591). IEEE Press.

Guzman, E., & Maalej, W. (2014, Aug). How do users like this feature? a fine grained sentiment analysis of app reviews. In *Proceedings of the 22nd ieee international requirements*

engineering conference (p. 153-162). IEEE.

Ha, E., & Wagner, D. (2013, Jan). Do android users write about electric sheep? examining consumer reviews in google play. In *Proceedings of the 10th ieee consumer communications and networking conference* (p. 149-157). IEEE.

Harman, M., Jia, Y., & Zhang, Y. (2012). App store mining and analysis: Msr for app stores. In *Proceedings of the 9th ieee working conference on mining software repositories* (pp. 108–111). IEEE Press.

Hoon, L., Vasa, R., Schneider, J.-G., & Mouzakis, K. (2012). A preliminary analysis of vocabulary in mobile app user reviews. In *Proceedings of the 24th australian computer-human interaction conference* (pp. 245–248). ACM.

Jha, A. K., Lee, S., & Lee, W. J. (2017). Developer mistakes in writing android manifests: An empirical study of configuration errors. In *Proceedings of the 14th international conference on mining software repositories* (pp. 25–36). Piscataway, NJ, USA: IEEE Press.

Karim, M. Y., Kagdi, H., & Penta, M. D. (2016, March). Mining android apps to recommend permissions. In *Proceedings of the 23rd ieee international conference on software analysis, evolution, and reengineering* (Vol. 1, p. 427-437). IEEE Press.

Keertipati, S., Savarimuthu, B. T. R., & Licorish, S. A. (2016). Approaches for prioritizing feature improvements extracted from app reviews. In *Proceedings of the 20th international conference on evaluation and assessment in software engineering* (pp. 33:1–33:6). ACM.

Khalid, H., Nagappan, M., Shihab, E., & Hassan, A. E. (2014). Prioritizing devices to test your app on: A case study of android game apps. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering* (pp. 610–620). IEEE.

Khalid, H., Shihab, E., Nagappan, M., & Hassan, A. E. (2015). What do mobile app users complain about? *IEEE Software*, *32*(3), 70–77.

Korner, J., Hitzges, L., & Gehrke, D. (2016). *Goko.* http://goko.me.

Lyons, K. (2015). What can a dumb watch teach a smartwatch?: Informing the design of smartwatches. In *Proceedings of the 2015 acm international symposium on wearable computers* (pp. 3–10). ACM.

Martin, W., Harman, M., Jia, Y., Sarro, F., & Zhang, Y. (2015). The app sampling problem for app

store mining. In *Proceedings of the 12th ieee/acm working conference on mining software repositories* (pp. 123–133). IEEE.

Martin, W., Sarro, F., Jia, Y., Zhang, Y., & Harman, M. (2017, Sept). A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, *43*(9), 817-847.

McIlroy, S., Ali, N., Khalid, H., & Hassan, A. E. (2016). Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. *Empirical Software Engineering*, *21*(3), 1067–1106.

McIlroy, S., Shang, W., Ali, N., & Hassan, A. (2015). Is it worth responding to reviews? a case study of the top free apps in the google play store. *IEEE Software*.

Min, C., Kang, S., Yoo, C., Cha, J., Choi, S., Oh, Y., & Song, J. (2015). Exploring current practices for battery use and management of smartwatches. In *Proceedings of the acm international symposium on wearable computers* (pp. 11–18). ACM.

Mujahid, S. (2017). Detecting wearable app permission mismatches: A case study on android wear. In *Proceedings of the 11th joint meeting on foundations of software engineering* (pp. 1065–1067). ACM.

Mujahid, S., Sierra, G., Abdalkareem, R., Shihab, E., & Shang, W. (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* (pp. 96–99). Piscataway, NJ, USA: IEEE Press.

Nagappan, M., & Shihab, E. (2016). Future trends in software engineering research for mobile apps. In *Proceedings of the 23rd ieee international conference on software analysis, evolution, and reengineering*. IEEE.

Pagano, D., & Maalej, W. (2013). User feedback in the appstore: An empirical study. In *Proceedings of the 21st ieee international requirements engineering conference* (p. 125-134). IEEE Press.

Palomba, F., Linares-Vsquez, M., Bavota, G., Oliveto, R., Penta, M. D., Poshyvanyk, D., & Lucia, A. D. (2015, Sept). User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *Proceedings of the 31st ieee international conference on software maintenance and evolution* (p. 291-300). IEEE.

Palomba, F., Salza, P., Ciurumelea, A., Panichella, S., Gall, H., Ferrucci, F., & De Lucia, A. (2017). Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the 39th international conference on software engineering* (pp. 106–117). IEEE Press.

Pandita, R., Xiao, X., Yang, W., Enck, W., & Xie, T. (2013). Whyper: Towards automating risk assessment of mobile applications. In *Proceedings of the 22nd usenix conference on security* (pp. 527–542). USENIX Association.

Panichella, S., Di Sorbo, A., Guzman, E., Visaggio, C. A., Canfora, G., & Gall, H. C. (2016). Ardoc: App reviews development oriented classifier. In *Proceedings of the 24th acm sigsoft international symposium on foundations of software engineering* (pp. 1023–1027). ACM.

Panichella, S., Sorbo, A. D., Guzman, E., Visaggio, C. A., Canfora, G., & Gall, H. C. (2015, Sept). How can i improve my app? classifying user reviews for software maintenance and evolution. In *Proceedings of the 31st ieee international conference on software maintenance and evolution* (p. 281-290). IEEE.

Park, S., & Jayaraman, S. (2003). Smart textiles: Wearable electronic systems. *MRS Bulletin*, *28*(8), 585591.

Rawassizadeh, R., Price, B. A., & Petre, M. (2015). Wearables: has the age of smartwatches finally arrived? *Communications of the ACM*, *58*(1), 45–47.

Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering (IST)*, *25*(4), 557–572.

Stevens, R., Ganz, J., Filkov, V., Devanbu, P., & Chen, H. (2013, May). Asking for (and about) permissions used by android apps. In *Proceedings of the 10th working conference on mining software repositories* (p. 31-40). IEEE Press.

Tehrani, K., & Michael, A. (2014, March). Wearable technology and wearable devices: Everything you need to know. *Wearable Devices Magazine*. Retrieved from http://www.wearabledevices.com/what-is-a-wearable-device/ (Accessed on August 25, 2017)

Teng, X. F., Zhang, Y. T., Poon, C. C. Y., & Bonato, P. (2008). Wearable medical systems for p-health. *IEEE Reviews in Biomedical Engineering*, *1*, 62-74.

Thelwall, M., Buckley, K., Paltoglou, G., Cai, D., & Kappas, A. (2010). Sentiment strength detection in short informal text. *Journal of the American Society for Information Science and Technology*, *61*(12), 2544–2558.

Tumbleson, C., & Winiewski, R. (2017). *Apktool - a tool for reverse engineering 3rd party, closed, binary android apps.* https://ibotpeaches.github.io/Apktool/. (Accessed on May 4, 2017)

Usman, M., Britto, R., Brstler, J., & Mendes, E. (2017). Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. *Information and Software Technology*, *85*(Supplement C), 43 - 59.

Vasa, R., Hoon, L., Mouzakis, K., & Noguchi, A. (2012). A preliminary analysis of mobile app user reviews. In *Proceedings of the 24th australian computer-human interaction conference* (pp. 241–244). ACM.

Watanabe, T., Akiyama, M., Sakai, T., & Mori, T. (2015). 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* (pp. 241–255). USENIX.

Wearable Software. (2016). *Android wear center.* http://www.androidwearcenter.com.

Wei, J. (2014, July). How wearables intersect with the cloud and the internet of things : Considerations for the developers of wearables. *IEEE Consumer Electronics Magazine*, *3*(3), 53-56.

Wei, X., Gomez, L., Neamtiu, I., & Faloutsos, M. (2012). Permission evolution in the android ecosystem. In *Proceedings of the 28th annual computer security applications conference* (pp. 31–40). ACM.

Wright, R., & Keith, L. (2014). Wearable technology: If the tech fits, wear it. *Journal of Electronic Resources in Medical Libraries*, *11*(4), 204-216.

Zhang, H., & Rountev, A. (2017). Analysis and testing of notifications in android wear applications. In *Proceedings of the 39th international conference on software engineering*. IEEE Press.