An empirical study of configuration changes and adoption in Android apps

Ajay Kumar Jha, Sunghee Lee and Woo Jin Lee
School of Computer Science & Engineering, Kyungpook National University
Daegu, Republic of Korea
ajaykjha123@yahoo.com, lee3229910@gmail.com, woojin@knu.ac.kr

Abstract: Android platform is evolving rapidly. Therefore, evolution and maintenance of Android apps are major concerns among developers. One of the essential components of each app is an Android manifest file, which is a configuration file used to declare various key attributes of apps. This paper presents an empirical study to understand app evolution through configuration changes. The results of this study will help developers in identifying change-proneness attributes, including change patterns and the reason behind the change, understanding the adoption of different attributes introduced in different versions of the Android platform, and understanding effort distribution pattern in configuration changes and taking proactive measures to reduce the effort. In this paper, we use a data mining approach. We analyze commit histories of Android manifest files of 908 apps to understand the app evolution. The results of this study show that most of the apps extend core functionalities and improve user interface over time, configuration changes are mostly influenced by functionalities extension, platform evolution, and bug reports, very few numbers of existing apps adopt new attributes introduced by the platform, apps are generally slow in adopting new attributes, and significant effort is wasted in changing configuration and then reverting back the change.

Keywords: Android app evolution; app maintenance; configuration changes; effort estimation; permission evolution

1. Introduction

Android is the most popular platform for mobile apps with more than 2.5 million apps available for download in the Google Play store (AppBrain, 2018). The platform is evolving rapidly. In a very short duration, it has evolved to API level 28 (SDK Platform release notes, 2018). One of the major challenges for app developers in such a rapidly evolving environment is to adopt the changes introduced by the platform in a timely manner (Joorabchi et al., 2013); otherwise, apps may have serious quality, reliability, or privacy issues. For example, Android has introduced an auto backup feature starting from API level 23, which automatically backs up user data by default, leading to privacy concerns (Panda, 2016, XDA Developers, 2015). Developers have to explicitly disable the feature in the Android manifest file (App Manifest Overview, 2018) to stop the auto backup. However, a study (McDonnell et al., 2013) on Android API evolution and its adoption by apps shows that the apps are not catching up with the pace of API evolution. Similarly, Li et al. (2018a) found that most deprecated APIs are still accessed by app code via popular libraries. Moreover, a study (Alharbi and Yeh, 2015) on design pattern changes in Android apps shows that the adoption rate of newly introduced design patterns is relatively low.

Another major challenge in the rapidly evolving environment is a short release cycle. Although frequently-updated apps are highly ranked by users (Mcllroy et al., 2016) and have benefits in app store ranking (Lim and Bentley 2013, Lynch 2012), a short release cycle may affect developers in properly planning the app release with additional functionalities. For example, we observe in this study that a large number of app components (activities, services, receivers, and providers) used to implement app functionalities are immediately removed after addition, resulting in wasted effort. Moreover, continuous changes in app functionalities may also result in poor quality apps (Hecht et al., 2015), especially when developers do not follow the well-structured app evolution strategy. A study (Mcllroy et al., 2016) shows that new features or functionalities are one of the major reasons behind frequent app updates. Moreover, a qualitative study (Joorabchi et al., 2013) reveals that requirements for mobile apps change rapidly and very often over time. Therefore, app maintenance is quite a challenging task for app developers (Minelli and Lanza, 2013). Furthermore, although recent advancement in mobile app development process (Jabangwe et al., 2018), particularly agile-based methods (Flora et al., 2014a), Android app developers hardly use any standard app development process (Inukollu et al., 2014, Dehlinger and Dixon, 2011, Wasserman, 2010, Jabangwe et al., 2018, Syer et al., 2013). They often rely on the best practices emerged from the historical evidence (Wasserman, 2010, Flora et al., 2014b), which is yet another major challenge in the rapidly evolving environment. There is a lack of access to historical data (Nagappan and Shihab, 2016). Although Android documentation provides some of the best practices for app development (Best practices, 2018), developers have to largely rely on developers' discussion forum such as Stack Overflow to get key insights (Rosen and Shihab, 2016, Linares-Vásquez et al., 2013a, Beyer and Pinzger, 2014, Guerrouj et al., 2015). Therefore, developers need to understand the app evolution pattern to proactively resolve various app maintenance challenges imposed by the rapidly evolving environment.

Several researchers have studied the evolution of Android apps. However, their subject is largely limited to API evolution and its impact on Android apps (McDonnell, 2013, Linares-Vásquez et al. 2013, Li et al., 2016, Li et al., 2018a) and permission evolution and its usage (Wei et al., 2012, Taylor and Martinovic, 2017, Calciati and Gorla, 2017). Researchers have given significant importance to API evolution because misuse of APIs can cause serious reliability and compatibility issues (Jha et al., 2019, Fan et al., 2018, Li et al., 2018b, He et al., 2018). Researchers have also given significant importance to permission evolution because it is used to implement security of Android apps (Enck et al., 2011), which is one of the major concerns among app users and developers (Rakestraw et al., 2013, Jha and Lee, 2016). Moreover, it has gone through several major changes, including migration from install-time permission to runtime permission (Permissions overview, 2018). In addition to permission, the Android manifest file (App Manifest Overview, 2018) is used to declare various key attributes of Android apps such as app components, app navigation, and device orientation (App Manifest Elements, 2018). In the context of app evolution, these attributes have been largely ignored by the researchers, despite the fact that some of these attributes represent the core functionalities of Android apps. Moreover, since the launch of the Android platform, several new attributes have been introduced in the Android manifest file. The impact of these changes on Android apps has not been established yet. Overall, our main motivation for this study is to get key insights about app maintenance and evolution patterns that can help practitioners in improving app development activities by taking proactive measures. Moreover, the results of this study can be valuable for future research contributions in the direction of app maintenance and evolution.

In this paper, we present an empirical study to understand app evolution through configuration changes. The configuration represents various elements and attributes declared in the Android manifest file (App Manifest Elements, 2018), including app permission. We analyze commit histories of Android manifest files of 908 free and open source Android apps to investigate various issues related to app maintenance and evolution. Specifically, we investigate frequently changed attributes and the reason behind the change, adoption of different attributes introduced by the Android platform, and developers' effort distribution in configuration changes. Although limited by the attributes declared in the Android manifest file, the study provides key insights on overall app evolution and maintenance. The study can also play an important role in regression test selection for Android apps (Do et al., 2016, Li et al., 2017, Chang et al., 2018). Existing studies (Kim and Porter, 2002, Li et al., 2007, Saha et al., 2015) have shown that regression test selection based on historical data is highly effective. To the best of our knowledge, this is the first empirical study that provides a quantitative analysis of configuration changes in Android apps. The major contributions of this paper can be summarized as follows:

- Performs a large-scale empirical study on configuration changes by mining commit histories of Android manifest files of 908 free and open source Android apps.
- Performs quantitative analysis of 12,341 commit histories of Android manifest files to:
    - Identify change-proneness attributes and the reason behind the change. The result shows that most of the apps extend core functionalities and improve user interface over time. The changes are mostly influenced by functionalities extension, platform evolution, and bug reports.

- Understand the adoption of different attributes introduced in different versions of the Android platform. The result shows that the percentage of existing apps adopting the newly introduced attributes or elements is very low. The result also shows that the adoption lag time is significantly high. However, it is decreasing over time.
- Understand developers' effort distribution in configuration changes. The result shows that significant effort is wasted in changing configuration and then reverting back the change.

The rest of the paper is organized as follows. Section 2 presents background on configuration changes in Android apps by providing an overview of the Android manifest file and defining the notion of configuration changes used by this paper. Section 3 presents related works. Section 4 presents the research design, including the methodology of obtaining dataset and analyzing commit histories. Section 5 presents the empirical results of configuration changes. The empirical results are discussed in Section 6, while threats that could affect the validity of the reported results are discussed in Section 7. Finally, the paper concludes in Section 8.

2. Background on configuration changes in Android apps

This section provides a brief overview of the Android manifest file (App Manifest Overview, 2018) and defines the notion of configuration changes used by this paper.

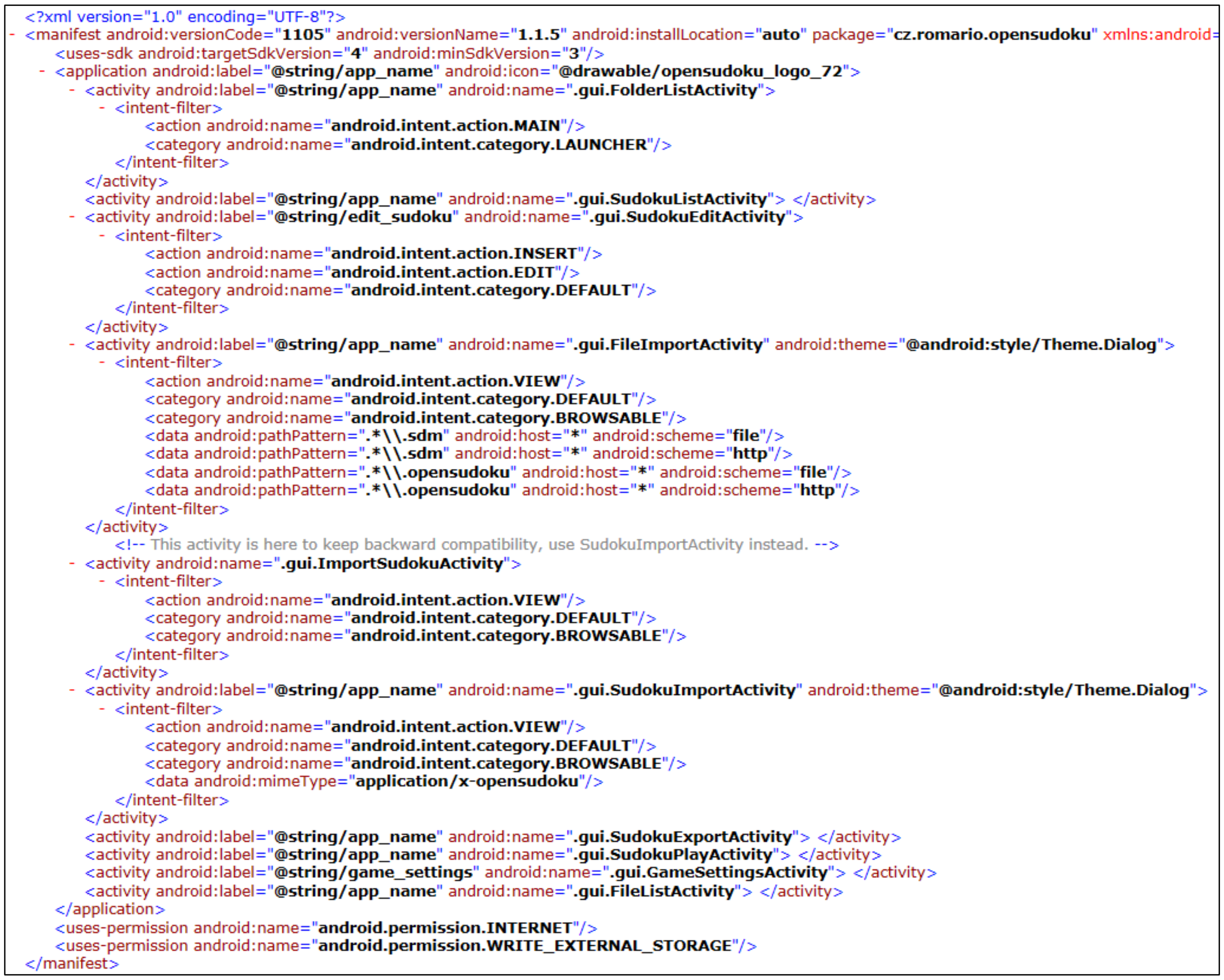
2.1 An overview of the Android manifest file

Each app must have an Android manifest file (App Manifest Overview, 2018), which is manually written by developers in XML format. Developers declare various elements and attributes in the Android manifest file that describes essential information about the app. The information in the Android manifest file is utilized not only by the app execution environment but also by external entities such as Google Play. Among various elements and attributes declared in the Android manifest file, we discuss some of the key elements and attributes in this section. Interested readers can find the complete description of all the elements and attributes of the Android manifest file in the Android documentation (App Manifest Elements, 2018). A real example of the Android manifest file of OpenSudoku app is shown in Fig. 1.

Components are the building blocks of Android apps. Features or functionalities in Android apps are implemented through the components. Android apps can be built using four types of components: activities, services, broadcast receivers, and content providers, which are declared in the Android manifest file using <activity>, <service>, <receiver>, and <provider> elements, respectively. The components of an app, which the system can launch, must be declared in the Android manifest file. As shown in Fig. 1, OpenSudoku app is built using ten activity components. Each type of component has a distinct functionality in Android apps. An activity represents a single user screen, a service is used to handle long-running tasks, a broadcast receiver handles broadcast events, and a content provider manages access to data. The properties of the components can be declared using various attributes of the component elements. For example, each component must have a name attribute. The value of the name attribute represents the name of the class that implements the component. Moreover, activity components can support various device orientations such as portrait and landscape by declaring screenOrientation attribute in <activity> elements.

In addition to defining properties of each component through the component elements, developers can define several app properties using various elements and attributes of the Android manifest file, mainly through <manifest> and <application> elements. Each app must define a package name in the Android manifest file, which is generally used as the unique app identifier on the system and Google Play. For example, OpenSudoku app has the package name "cz.romario.opensudoku" declared in the <manifest> element. Some attributes can be defined for both component and app such as icon, theme, and navigation. An icon represents an icon for the app and the default icon for each of the app's components. A theme is a type of style that is applied to an entire app or an activity. The navigation property defines how users can navigate between screens or activities. There are several other properties that can be defined for components and apps (App Manifest Elements, 2018).

Security is a critical aspect of Android apps. Android uses permission-based security (Permissions overview, 2018, Jha and Lee, 2016) at the application level. Permissions are used to protect the sensitive system and app resources. They are defined, declared, and enforced through various elements and attributes of the Android manifest file. Sensitive system resources such as location are protected with pre-defined system permissions; whereas, sensitive app resources are protected by defining custom permissions in the Android manifest file via <permission> elements. Apps willing to access the protected system or app resources must declare permissions in the Android manifest file via <uses-permission> or <uses-permission-sdk-23> elements. For example, OpenSudoku app does not define any custom permissions. However, it uses INTERNET and WRITE_EXTERNAL_STORAGE system permissions to access internet and external storage media, respectively.



```
  <?xml version="1.0" encoding="UTF-8"?>
- <manifest android:versionCode="1105" android:versionName="1.1.5" android:installLocation="auto" package="cz.romario.opensudoku" xmlns:android=
     <uses-sdk android:targetSdkVersion="4" android:minSdkVersion="3"/>
   - <application android:label="@string/app_name" android:icon="@drawable/opensudoku_logo_72">
      - <activity android:label="@string/app_name" android:name=".gui.FolderListActivity">
         - <intent-filter>
              <action android:name="android.intent.action.MAIN"/>
              <category android:name="android.intent.category.LAUNCHER"/>
           </intent-filter>
        </activity>
        <activity android:label="@string/app_name" android:name=".gui.SudokuListActivity"> </activity>
      - <activity android:label="@string/edit_sudoku" android:name=".gui.SudokuEditActivity">
         - <intent-filter>
              <action android:name="android.intent.action.INSERT"/>
              <action android:name="android.intent.action.EDIT"/>
              <category android:name="android.intent.category.DEFAULT"/>
           </intent-filter>
        </activity>
      - <activity android:label="@string/app_name" android:name=".gui.FileImportActivity" android:theme="@android:style/Theme.Dialog">
         - <intent-filter>
              <action android:name="android.intent.action.VIEW"/>
              <category android:name="android.intent.category.DEFAULT"/>
              <category android:name="android.intent.category.BROWSABLE"/>
              <data android:pathPattern=".*\\.sdm" android:host="*" android:scheme="file"/>
              <data android:pathPattern=".*\\.sdm" android:host="*" android:scheme="http"/>
              <data android:pathPattern=".*\\.opensudoku" android:host="*" android:scheme="file"/>
              <data android:pathPattern=".*\\.opensudoku" android:host="*" android:scheme="http"/>
           </intent-filter>
        </activity>
           <!-- This activity is here to keep backward compatibility, use SudokuImportActivity instead. -->
      - <activity android:name=".gui.ImportSudokuActivity">
         - <intent-filter>
              <action android:name="android.intent.action.VIEW"/>
              <category android:name="android.intent.category.DEFAULT"/>
              <category android:name="android.intent.category.BROWSABLE"/>
           </intent-filter>
        </activity>
      - <activity android:label="@string/app_name" android:name=".gui.SudokuImportActivity" android:theme="@android:style/Theme.Dialog">
         - <intent-filter>
              <action android:name="android.intent.action.VIEW"/>
              <category android:name="android.intent.category.DEFAULT"/>
              <category android:name="android.intent.category.BROWSABLE"/>
              <data android:mimeType="application/x-opensudoku"/>
           </intent-filter>
        </activity>
        <activity android:label="@string/app_name" android:name=".gui.SudokuExportActivity"> </activity>
        <activity android:label="@string/app_name" android:name=".gui.SudokuPlayActivity"> </activity>
        <activity android:label="@string/game_settings" android:name=".gui.GameSettingsActivity"> </activity>
        <activity android:label="@string/app_name" android:name=".gui.FileListActivity"> </activity>
     </application>
     <uses-permission android:name="android.permission.INTERNET"/>
     <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
  </manifest>
```

Fig. 1. Android manifest file of OpenSudoku app.

The importance of the Android manifest file can be perceived by the type of information it contains. In addition to quality attributes such as theme and icon, the Android manifest file contains core functional attributes and security attributes. The quality attributes such as app name, theme, and icons are critical factors that influence users' choice of apps (Lim et al., 2015). Therefore, the study of the Android manifest file can reveal several critical aspects of Android apps. In this study, we analyze the commit histories of the Android manifest file to understand how the properties of Android apps represented by the elements and their attributes of the Android manifest file evolve over time. For example, if the components such as activities, services, broadcast receivers, or content providers are added in the Android manifest file through a commit, we document that the functionalities of the app have been extended. Similarly, if the components are removed, we document that the existing functionalities of the app have been removed.

2.2 Configuration changes

The term configuration can be used broadly for customizing and adapting an app as well as its execution environment to different users' requirements and contexts (Sayagh et al., 2018). Moreover, as described by Sayagh et al. (2018), a configuration option of an app corresponds to a key-value pair where a key represents a configuration option's name and its value represents the choice by a practitioner or user regarding that option for a specific instance of the system.

If we consider the aforementioned definition of configuration, in addition to the Android manifest file, there are several other configuration files in Android apps such as various resource files. However, given the importance of the Android manifest file in Android research community and the variety of essential information it contains, we limit our study to the Android manifest file. In this paper, configuration represents various elements and attributes declared in the Android manifest file. Therefore, the term configuration change refers to each change in the elements or attributes of the Android manifest file since the file was originally created. In this paper, we evaluate the configuration change at commit-level. Therefore, if $AMF_{original}$ is the original Android manifest file and $AMF_{commit(n)}$ is the nth commit on the Android manifest file, configuration change of the app is represented by $CC_{app} = (AMF_{original} - AMF_{commit(1)}) + (AMF_{commit(1)} - AMF_{commit(2)}) + \ldots + (AMF_{commit(n-1)} - AMF_{commit(n)})$. For example, if the Android manifest file shown in Fig. 1 represents the $AMF_{original}$ for the OpenSudoku app and the developer of the app added an activity component, a permission, and an app theme in three different commits, the configuration change for the app is represented by the sum of the three different commits. Therefore, $CC_{OpenSudoku}$ = activity + permission + app theme.

3. Related works

Researchers have performed several studies on Android app evolution. Particularly, permission evolution and its usage in Android apps, API evolution and its effect on Android apps, and app reviews analysis for maintenance and evolution of Android apps. This section presents a literature review on the state of the art in each category.

3.1 Permission evolution and its usage in Android apps

Wei et al. (2012) performed a study on permission evolution and its usages, targeting the entire Android ecosystem. They made some key observations such as a dangerous-level set of permissions is the most frequent and continues to grow, Android platform permissions are not becoming fine-grained, and apps are violating the principle of least privilege. However, the permission system in Android has been substantially changed since Wei et al. (2012) performed their study. Taylor and Martinovic (2017) performed a longitudinal study of app permission usage across the Play store, exclusively targeting dangerous permissions. One of their key observation is that free apps and popular apps are more likely to ask for additional permissions. Calciati and Gorla (2017) performed a study on permission evolution to understand permission request change and their usage. Their observations do not significantly deviate from the existing studies (Wei et al., 2012, Taylor and Martinovic, 2017). However, they made some additional observations such as developers are requesting permissions that fall in the same group and are implicitly granted. In contrast to these studies, our study is significantly broader. We investigate all kinds of configuration changes and their usage in Android apps, including permission.

3.2 API evolution and its effect on Android apps

McDonnell et al. (2013) studied the evolution of Android APIs and its impact on Android apps. They analyzed 10 open source apps to study the impact. They investigated various factors including time taken for apps to adopt new APIs and the relationship between API usage and adoption. Their results show that the app adoption is not catching up with the pace of API evolution and fast evolving APIs are used more by apps than slow evolving APIs but the average time taken to adopt new version is longer for fast-evolving APIs. Linares-Vásquez et al. (2013) performed an empirical study to investigate the impact of API fault- and change-proneness on the success of Android apps. The success was estimated from user ratings. Their results show that APIs used by successful apps are on average significantly less fault-prone than APIs used by unsuccessful apps. Furthermore, APIs used by successful apps are on average less change-prone than APIs used by unsuccessful apps. Li et al. (2016) performed an empirical study to understand how inaccessible (internal or hidden) APIs evolve over time and who uses them. In addition to various findings, they found that the turn-over of inaccessible APIs is very high, most of them being removed from the framework after a few version updates. However, they observed that relatively less number of Android apps (5.4%) adopted the inaccessible APIs. In contrast to these studies, we investigate attributes evolution in the Android manifest files and their impact on Android apps. However, we observe in this study that apps are slow in adopting attributes introduced by the platform, which shows a similar pattern with APIs adoption observed by McDonnell et al. (2013).

3.3 Design pattern evolution in Android apps

Alharbi and Yeh (2015) presented a data-mining approach to analyze design pattern changes in Android apps. Moreover, they analyzed more than 24k apps over a period of 18 months to understand various design pattern changes. They also presented the result of the eight most illustrative design pattern changes in Android apps. Unlike their study, we primarily focus on configuration changes patterns. However, some elements of the Android manifest file represent design elements such as app navigation. Moreover, similar to their result on design pattern adoption rate, we found that the adoption rate of newly introduced attributes is relatively low.

3.4 App evolution considering various parameters

In addition to specific issues such as API evolution, some researchers (Calciati et al., 2018, Tian et al., 2015, Nayebi et al., 2018) have considered various parameters to study overall app evolution. Calciati et al. (2018) proposed a framework to analyze the evolution of Android apps. The framework extracts and visualizes various information such as network traffic and uses of sensitive data and libraries, and combines them to create a comprehensive report on the evolution of the analyzed apps. They also performed an empirical study, which shows that apps tend to have more leaks of sensitive data over time and that the majority of API calls relative to dangerous permissions are added to the code in releases posterior to the one where the corresponding permission was requested. Tian et al. (2015) investigated 28 factors such as app size, code complexity, and library dependency to understand characteristics of successful apps. They chose app rating as a proxy for app success. They found that high-rated apps are statistically significantly different from low-rated apps in 17 out of the 28 factors. The most closely related work was performed by Nayebi et al. (2018). They analyzed commit histories of Android apps and performed a survey with mobile app developers to understand the deletion of app functionality. They observed that deletion of app functionality is common in mobile apps and app functionalities are deleted mainly to fix bugs, to maintain compatibility, or to improve the user experience. In contrast to the study performed by Nayebi et al. (2018), we mainly investigate configuration changes in Android apps, where deletion of app functionality is one of the various reasons behind configuration changes.

3.5 App reviews analysis for maintenance and evolution of Android apps

In addition to the source code analysis, researchers have proposed various tools and techniques (Galvis-Carreno and Winbladh, 2013, Carreño and Winbladh, 2013, Fu et al., 2013, Guzman and Maalej, 2014, Chen et al., 2014, Maalej and Nabil, 2015, Palomba et al., 2017, Palomba et al., 2018, Sorbo et al., 2016, Panichella et al., 2015, Panichella et al., 2016, Genc-Nayebi and Abran, 2017) to extract information from user reviews useful for maintaining and evolving mobile apps. Although we do not analyze user reviews in this study, these studies provide key insights on requirement elicitation for mobile apps, which directly influence configuration changes. However, we perform limited analysis of app ratings in this study.

3.6 Empirical studies on Android manifests

Researchers have studied Android manifests for various purposes. Barrera et al. (2010) presented a methodology for the empirical analysis of permission-based security using Self-Organizing Map. They used the methodology to analyze the strength and weakness of the Android permission model on more than 1k apps. We developed a static analysis tool (Jha et al., 2017) to detect errors in the Android manifest file and then performed an empirical study on configuration errors. We found more than 59k configuration errors in more than 11k apps. In another empirical study (Jha and Lee, 2018), we investigated inter-app communication and its security risk by analyzing Android manifest files of more than 13k apps. The result shows that a large number of apps (65%) perform inter-app communication and 54% of the apps are at serious security risk. In contrast to these studies, the methodology and purpose of our study in this paper are completely different. We analyze commit histories of the Android manifest files to investigate various issues related to app evolution and maintenance.

4. Empirical study design

A data mining approach is used in this study to understand various factors involving configuration changes. Historical data from commit histories of the Android manifest files were manually extracted, which were then analyzed to understand configuration changes in Android apps. We chose to extract the historical data manually despite the fact that the manual process may lead to inaccuracies because we wanted to understand the purpose behind the commit by analyzing the developer's comment if available. Moreover, to the best of our knowledge, there are not any reliable tools that can automatically extract commit histories.

4.1 Data collection

We used F-Droid (2018), a repository of free and open source Android apps, to select apps for this empirical study. We collected URLs of all the apps stored on the F-Droid repository and selected only those apps that are hosted on GitHub. This criterion resulted in 1,029 apps. Then, we performed a filtering process on the selected apps. First, we removed 76 apps that had no commit histories in the Android manifest files. For the remaining 953 apps, we manually analyzed each commit of the Android manifest files. We further removed 45 apps that had only version and SDK related commits, resulting in a dataset of 908 apps with 20,793 commits. The commits do not include initial commits and merge commits. Among 20,793 commits in 908 apps, 7,757 commits are exclusively related to version and SDK update. However, we found that version and SDK are either already maintained through Gradle in most of the apps or developers have moved the version and SDK to Gradle in several apps at a later stage of the app development. Therefore, we have excluded the version and SDK commits from our further study, resulting in a total 13,036 commits in 908 apps. Moreover, we removed 479 commits from 285 apps that are exclusively related to formatting and syntax changes. We also removed 216 commits from 137 apps that are exclusively related to developers' comments, resulting in a final dataset of 12,341 commits in 908 apps. The dataset for this study was collected from October to January of 2017/18. We have made the dataset publically available for interested readers (Dataset, 2018).

Among the final dataset of 908 apps, 694 apps are available in the Google Play, which have various download ranges as shown in Fig. 2. We could not access the download range of 48 apps because of the Play store localization. Overall, our dataset has diverse categories of apps including apps from the Google Play and third-party app stores. The majority of apps (75.88%) in our dataset has 1 to 10 components at the time of project creation. In this study, we analyze the commit histories of the Android manifest files to obtain the results. Therefore, it is important to understand the timeline of the commits in each app. Fig. 3 shows the time since the first commit in the Android manifest files of each app. Clearly, the first commit in most of the apps is 1 to 3 years old.

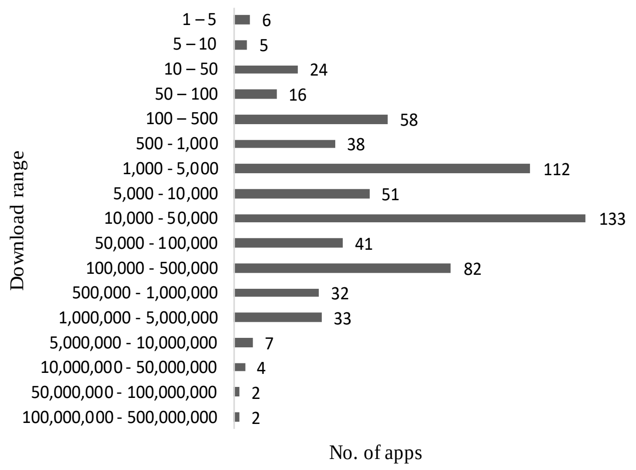


Fig. 2. App download range in Google Play store.

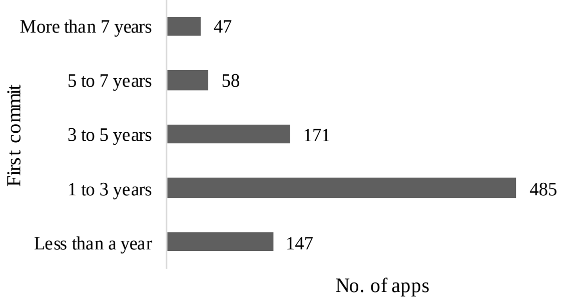


Fig. 3. Commit history timeline in apps.

4.2 Research method

For quantitative analysis of the commit histories, the first and the second author manually and independently went through each commit of the Android manifest files and recorded the changed code and developer's comment about the change. For example, if largeHeap="false" was removed and largeHeap="true" was added, the authors recorded the event as "*large heap enabled*". The GitHub view was used as a tool to analyze the changed code. In GitHub view, added or removed code are highlighted in different colors. Although the process is not prone to human bias, recording a large number of changed code may have human errors. Therefore, to mitigate human errors, the authors manually verified the recorded artifacts by cross-checking.

In addition to the code changed in a commit, developers usually provide a short comment explaining the change. Let us take an example of three different commits from three different real apps. In all the three commits, developers added largeHeap="true" in the Android manifest files. Given the current information, we can say that developers have added the code to enhance the performance of the apps, which provides the generic reason behind the configuration change. Now, let us see comments of the three commits: "*v4.5 (WIP)*", "*bitmap and scrolling*", and "*Use largeHeap to fix OOM errors when checking db and importing*". The first comment does not provide any information about the changed code. However, the second and third comments provide very specific reasons behind the need for performance enhancement. Primarily, our goal is to find generic reasons behind configuration changes. However, we use the comments in some parts of the paper to provide key insights behind configuration changes. Therefore, the first and second authors manually and independently analyzed the comments using an open coding approach (Seaman, 1999, Miles et al., 2013). Each author recorded the reason behind the configuration change by analyzing the comments and the changed code. The disagreement (8% of the comments) about the reason behind the configuration change was resolved through discussion between the first and second authors. The reason behind the configuration change was analyzed after cross-checking the recorded artifacts, that is after data collection. Therefore, the open coding approach was not compromised in the process.

4.3 Research questions

The main goal of this study is to understand app evolution through configuration changes. In this context, we designed the following three research questions:

RQ1: What are the most frequently changed attributes in Android apps? This research question serves as the main purpose of our study. We can get key insights on app evolution pattern by understanding which attributes or features of Android apps are changed (added or removed) over time and the reason behind the change. The answer to this research question will help app developers in proactively concentrating their development and maintenance effort in areas that are changed most frequently. For example, the result can help developers in regression test selection.

RQ2: What is the adoption pattern of new attributes introduced by the Android platform? The main purpose of this research question is to understand how existing Android apps adopt newly introduced attributes, specifically time-lag in adoption and frequency of adoption. The answer to this research question will help platform developers in identifying the attributes that may not be helpful for app developers or the attributes that are stagnant. Moreover, they can evaluate the overall current adoption practices and provide various ways to improve the adoption rate.

RQ3: What is the effort distribution pattern in configuration changes? The main purpose of this research question is to identify wasted effort by understanding the effort distribution pattern in configuration changes. The answer to this research question will help app developers in reducing the maintenance effort by taking proactive measures during app development.

5. Empirical results

In this section, we present empirical results of configuration changes in Android apps. Fig. 4 shows the top-10 most frequently changed attributes in Android apps. For example, app components have been added or removed from 696 apps using 4,459 commits. We have grouped the related configurations of the Android manifest file to represent an app attribute. We found 28 different app attributes changed in this study. In this paper, we present the results of the top-10 most frequently changed attributes.

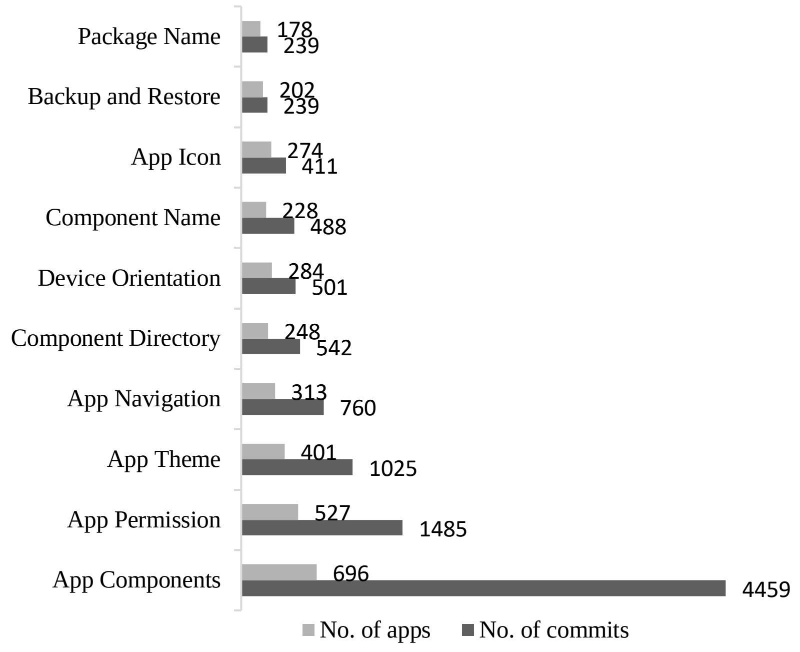


Fig. 4. Top-10 most frequently changed attributes.

5.1 App components

In the dataset, 4,625 components have been added to 659 apps; whereas, 1,601 components have been removed from 399 apps. Out of total 908 apps, components have been added and removed from 696 apps, which means 212 (23%) apps do not add or remove components over time. We further analyzed app properties such as project creation date, download range in the Play store if available, and app size of the 212 apps to find any correlations between app properties and component addition or removal. However, properties of the 212 apps do not significantly deviate from the properties of the other apps in the dataset. Martin et al. (2016) found that new features addition positively affect the success of the apps. Therefore, we investigated whether the stagnant apps that do not add or remove components or features are negatively affected. Similar to previous works (Linares-Vásquez et al., 2013, Martin et al., 2016), we used app ratings as a success metric. Moreover, we used app components as a proxy for app features (Nayebi et al., 2018). Out of 212 stagnant apps in the dataset, 153 apps are available in the Play store. We manually extracted app ratings of 153 apps from the apps' description page in the Play store. The maximum average rating for an app can be 5. Out of 153 apps, 19 apps do not have any ratings, 28 apps have ≥ 3.4 and < 4.0 average ratings, and 106 apps have ≥ 4.0 average ratings. The apps that do not have any ratings have also very low download ranges. Overall, most of the apps (69.3%) have ≥ 4.0 average ratings. Therefore, ***the apps that do not add new features over time are not negatively affected in their success***.

Fig. 5 shows the distribution of component changes in 696 apps: added components are marked positive and removed components are marked negative. The figure clearly shows that most of the apps add one to three components and remove one or two components over time. However, a small number of apps add and remove a significantly large number of components, reaching up to 104 and 34 components, respectively. We further analyzed 50 apps, selecting 25 apps from each set with the highest number of components added and removed. Unlike most of the apps (75.88%) with 1 to 10 components in the dataset, we observed that the majority of apps in this category are large-sized apps with more than 20 components at project creation time. Therefore, we investigated whether there is a correlation between component changes and number of components available at project creation time. We performed this investigation on large apps with more than 20 components because we have already established that app size generally does not affect components addition or removal. Out of 696 apps that change components, 79 apps have more than 20 components at the project creation time. We further filtered 6 outlier apps that have either more than 90 components at the project creation time or changed more than 90 components. Then, we calculated both Pearson's correlation and Spearman's correlation coefficient on 73 apps. The results of both Pearson's correlation (r=0.105, N=73, p=0.38) and

Spearman's correlation coefficient (r=0.111, N=73, p=0.35) indicate that there is no correlation between component changes and number of components available at project creation time. Therefore, ***new features addition or existing features removal does not depend on the number of features provided by the app.***

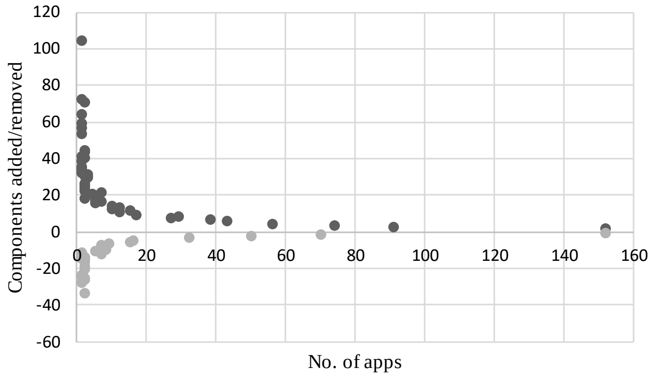


Fig. 5. Components added or removed from apps.

Fig. 6 shows the number of components added and removed according to their types. For example, 3,182 activities have been added to 593 apps and 1,148 activities have been removed from 340 apps. Clearly, activities are the most frequently changed components; whereas, providers are the least frequently changed components in Android apps. The result is not surprising because activities are the most frequently used and providers are the least frequently used components in Android apps (Jha and Lee, 2018).

We also investigated the macro pattern of component usage. Table 1 shows the macro pattern, where '0' represents the state in which an app does not use a particular component, '1' represents the state in which an app uses a particular component, and '→' represents a state transition. Clearly, component addition dominates the component usage with 69.62%. We observed that components have been added to extend the specific functionalities of the apps in most of the cases. However, there are some cases where components have been added to extend the general functionalities of the apps. For example, 368 activity components have been added to add a setting or about screen. A setting screen displays various preferences for app users; whereas, an about screen displays information about the app. Overall, ***new features addition significantly dominates other feature changes activities such as features removal.***

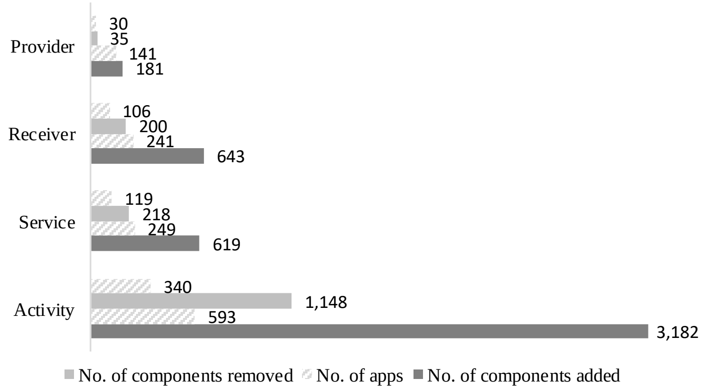


Fig. 6. Types of components added or removed from apps.

We also observed one interesting pattern with significant frequency (14.97%), addition and then removal of components. Out of total 1,601 components removed from the apps, more than 50% of the removed components were first added then removed from the apps. Moreover, a large number of components were removed very early after addition. For example, 164 components were removed in the next commit following the commit in which the components were added. In addition to the patterns shown in the table, we found four other patterns. However, the patterns are not included in the table due to their insignificant frequency. Overall, ***app developers spend a significant amount of time adding new features and then removing the added features.***

Table 1. Macro pattern of component addition or deletion.

Macro pattern	**Frequency**
$0 \rightarrow 1$	69.62%
$1 \rightarrow 0$	13.81%
$0 \rightarrow 1 \rightarrow 0$	14.97%
$0 \rightarrow 1 \rightarrow 0 \rightarrow 1$	0.92%

5.2 Component name

Developers must define a name for each component using a name attribute in the component elements of the Android manifest file. It represents the name of the class that implements the component, a subclass of Activity, Service, BroadcastReceiver, or ContentProvider. Android does not provide any standards to define the name of the components. However, developers generally follow Java class naming conventions (Naming Conventions, 2018). Moreover, they append component type at the end of the name.

In the dataset, 543 components' name including 403 activities, 76 services, 59 receivers, and 5 providers have been changed in 228 apps. We analyzed Java class files of the components to differentiate between addition/removal of the components and change in the components' name. If the Java class file has been removed or added then we considered that as removal or addition of the component; whereas, if only the name of the class file has been changed, we considered that as a change in the component's name. Among 543 components' name change, developers have replaced certain words with more suitable words in 171 cases, for example, screen and view were replaced by activity; whereas, the task was replaced by service. In 104 cases, developers have either appended the word Activity at the end of the name or they have changed the order of the word Activity in the name. Furthermore, developers have used the more descriptive name in 99 cases; whereas, they have shortened the name in 60 cases. Overall, ***developers have changed the name of the components to reflect their type and core functionalities.***

5.3 Component directory

The value of name attribute in the component can be a fully qualified class name or a shorthand declared by using a period as a first character in the name. In this study, we could not clearly observe any preferences between the fully qualified name and shorthand. However, we observed that developers have resolved some bugs by using the fully qualified name. One key factor which we observed is that developers have changed directories of the components in several apps. In the dataset, 1,157 components' directories including 852 activities, 137 services, 149 receivers, and 19 providers have been changed in 248 apps. ***In most of the cases, developers have created and used separate directories for each type of components. Moreover, they have renamed the directories in some cases to match the component types.***

5.4 App permission

In the dataset, system permissions have been changed (added or removed) in 527 apps. Fig. 7 shows the distribution of permission changes in 527 apps: added permissions are marked positive and removed permissions are marked negative. The figure clearly shows that most of the apps add one or two permissions over time; whereas, a small number of apps remove permissions over time, largely a single permission. However, in comparison to the result in Wei et al. (2012) work, the percentage of apps removing permissions over time is significantly large in our dataset. Table 2 shows the top-7 most frequently added and removed permissions in our dataset. The most frequently added permissions are mostly in-line with the result obtained by Wei et al. (2012). It is also consistent with the most frequently used permissions (Jha and Lee, 2018). However, none of the most frequently removed permissions match with the result obtained by Wei et al. Moreover, four permissions are in the list of both top-7 most frequently added and removed permissions.

The result clearly indicates a change in permission removal pattern since Wei et al. (2012) performed their study (2009-2011). First, the percentage of apps removing permissions over time is significantly large. Second, apps are not only removing permissions that do not improve user experience such as ACCESS_MOCK_LOCATION. We observed two main reasons for the change in permission removal pattern: platform-induced changes and developer sensitivity towards the use of permissions. Several system permissions have deprecated over time, for example, READ_OWNER_DATA and RECEIVE_MMS permissions listed in top-5 most frequently removed permissions in Wei et al. (2012) work are deprecated. Furthermore, the Android platform no longer uses some permissions starting with a specific version. For example, starting from API level 19, writing files in apps' private external storage do not require WRITE_EXTERNAL_STORAGE permission, which is the most frequently removed permission in our dataset. We also observed by analyzing developers' comments during permission change that developers have become more sensitive towards permissions request. In most of the cases, developers have explicitly mentioned that they are removing unused or unnecessary permissions. Furthermore, in several cases, they removed permissions by using techniques that avoid permissions request. For example, they used a FileProvider to avoid WRITE_EXTERNAL_STORAGE permission request. Overall, ***developers are increasingly avoiding the use of dangerous permissions and removing the unused permissions.***

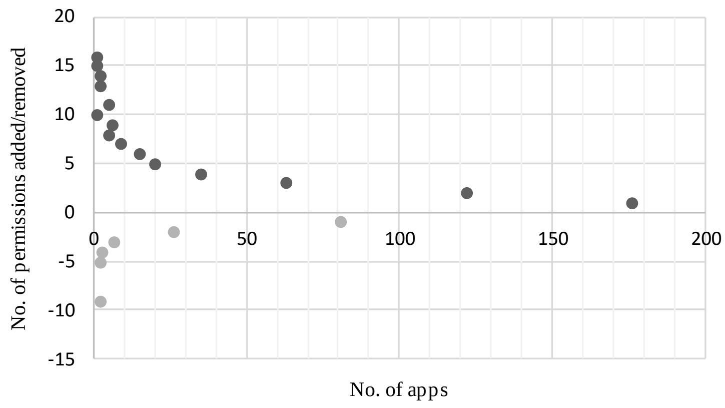


Fig. 7. Permission change in the apps.

Similar to Wei et al. (2012) and Calciati and Gorla (2017) works, we investigated the macro pattern of permission evolution. Table 3 compares results obtained by Wei et al. (2012) and Calciati and Gorla (2017) with ours, where '0' represents the state in which an app does not use a particular permission, '1' represents the state in which an app uses a particular permission, and '$\rightarrow$' represents a state transition. One key difference between their works (Wei et al., 2012, Calciati and Gorla, 2017) and ours is that we studied the permission evolution on commit-level; whereas, they studied the evolution on release-level. We can observe two interesting patterns from the results shown in Table 3: decline in permission addition and incline in permission removal since Wei et al. (2012) performed their study. Similar to the results obtained by Calciati and Gorla (2017), we obtained additional and longer patterns. The results shown in Table 3 account for 86.33% of patterns found in our study, with the remaining 13.67% containing the $0 \rightarrow 1 \rightarrow 0$ pattern (10.26%), the $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ pattern (2.87%), and four other longer patterns. ***Overall trends of decline in permission addition and incline in permission removal show that developers are improving their practices of permission usage.***

Table 2. Most frequently added and removed permissions.

Added Permissions	Removed Permissions
WRITE_EXTERNAL_STORAGE	WRITE_EXTERNAL_STORAGE
INTERNET	INTERNET
ACCESS_NETWORK_STATE	BILLING
RECEIVE_BOOT_COMPLETED	ACCESS_NETWORK_STATE
READ_EXTERNAL_STORAGE	WAKE_LOCK
WAKE_LOCK	READ_PHONE_STATE
VIBRATE	GET_ACCOUNTS

Similar to Wei et al. (2012) and Calciati and Gorla (2017) works, we investigated the macro pattern of permission evolution. Table 3 compares results obtained by Wei et al. (2012) and Calciati and Gorla (2017) with ours, where '0' represents the state in which an app does not use a particular permission, '1' represents the state in which an app uses a particular permission, and '$\rightarrow$' represents a state transition. One key difference between their works (Wei et al.,

2012, Calciati and Gorla, 2017) and ours is that we studied the permission evolution on commit-level; whereas, they studied the evolution on release-level. We can observe two interesting patterns from the results shown in Table 3: decline in permission addition and incline in permission removal since Wei et al. (2012) performed their study. Similar to the results obtained by Calciati and Gorla (2017), we obtained additional and longer patterns. The results shown in Table 3 account for 86.33% of patterns found in our study, with the remaining 13.67% containing the $0 \rightarrow 1 \rightarrow 0$ pattern (10.26%), the $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ pattern (2.87%), and four other longer patterns. ***Overall trends of decline in permission addition and incline in permission removal show that developers are improving their practices of permission usage.***

Unlike the existing works on permission evolution (Wei et al., 2012, Taylor and Martinovic, 2017, Calciati and Gorla, 2017), we also investigated custom permissions, which are used by developers to protect sensitive app resources that are not defined by the system. Unlike predefined system permissions, developers need to define custom permissions in their apps. In the dataset, 21 custom permissions have been added to 20 apps; whereas, 8 custom permissions have been removed from 8 apps. The result clearly indicates the limited use of custom permissions. App users are often suspicious of unknown permissions request, including custom permissions. Therefore, developers should avoid the use of custom permissions.

Table 3. Macro pattern of permission change.

Macro pattern	**Wei et al., (2012) Results**	**Calciati and Gorla, (2017) Results**	**Our results**
$0 \rightarrow 1$	90.46%	80.18%	84.64%
$1 \rightarrow 0$	8.59%	15.27%	13.18%
$1 \rightarrow 0 \rightarrow 1$	0.84%	3.27%	1.83%
$1 \rightarrow 0 \rightarrow 1 \rightarrow 0$	0.11%	1.27%	0.34%

5.5 App navigation

Consistent navigation is an essential element for overall user experience. Developers can manage app navigation by declaring certain attributes in the Android manifest file. In this study, we identified 14 attributes that have been used to change app navigation of 313 apps. The attributes are shown in Table 4. Developers have used a combination of attributes to change the app navigation. However, the most frequently used attributes are parentActivityName and launchMode, which we will discuss in detail.

Back Navigation: A task is a collection of activities that users interact with when performing a certain job. The activities are arranged in a back stack. The system maintains the back stack of activities while the user navigates an app, which allows the system to navigate backward when the user presses the Back button. The system manages tasks and the back stack by placing all the activities started in succession in the same task and last in, first out order, respectively. However, developers may need to customize the default behavior. While most of the attributes shown in Table 4 are used to customize the default behavior, the main attribute is launchMode.

Launch mode specifies how an activity should associate with a task when it is launched. Activities can be launched in four different modes: standard, singleTop, singleTask, and singleInstance. The default mode is standard in which the system creates a new instance of the activity in the task from which it was started. In singleTop launch mode, an instance of the activity is only created if an instance of the activity does not already exist at the top of the current task. In both standard and singleTop launch modes, an activity can be instantiated multiple times, each instance can belong to different tasks, and one task can have multiple instances. In contrast to standard and singleTop launch modes, only one instance of an activity can exist at a time if the activity is launched in singleTask or singleInstance launch modes. In both singleTask and singleInstance launch modes, the system creates a new task and instantiates the activity at the root of the new task. However, the system does not launch any other activities into the task holding the instance in singleInstance launch mode.

In the dataset, singleTop, singleTask, or singleInstance launch modes have been added to 190 activities of 134 apps. It suggests that ***default navigation behavior is not sufficient in several instances***. Moreover, Android documentation states that singleTask and singleInstance launch modes are not appropriate for apps because they result in an interaction model that is likely to be unfamiliar to users and is very different from most other apps (Interaction model, 2018). However, developers have added singleTask or singleInstance launch modes in 108 activities of 70 apps. Furthermore, they have changed launch modes to singleTask or singleInstance in 74 activities of 36 apps. It indicates that ***developers are indeed using the app navigation model that is likely to be unfamiliar to users***. We also found that developers have removed or changed singleTask or singleInstance launch modes in 41 activities of 27 apps. However, in most of the cases, they have changed or removed the launch modes to fix bugs. In only two cases, developers have explicitly mentioned that they have made the changes to fix unfamiliar app navigation. If a user leaves a task for a long time, the system clears the task of all activities except the root activity. However, the default behavior can be changed by using alwaysRetainTaskState, clearTaskOnLaunch, and finishOnTaskLaunch attributes. Developers have used the attributes to change the default behavior in 42 activities of 21 apps.

Table 4. Attributes used in app navigation.

Attribute Name	**Declaring Element(s)**
allowTaskReparenting	<activity>, <application>
alwaysRetainTaskState	<activity>
autoRemoveFromRecents	<activity>
clearTaskOnLaunch	<activity>
documentLaunchMode	<activity>
excludeFromRecents	<activity>
finishOnTaskLaunch	<activity>
launchMode	<activity>
maxRecents	<activity>
noHistory	<activity>
parentActivityName	<activity>
support.PARENT_ACTIVITY	<meta-data>
taskAffinity	<activity>, <application>
uiOptions	<activity>, <application>

Developers have added then removed launch modes and vice-versa in 37 activities of 29 apps. They have also changed the launch modes and then reverted back the change in 19 activities of 11 apps. We observed that they were trying to fix navigation issues by making some changes and then reverting back the changes. It suggests that ***developers are facing difficulties in customizing the default navigation behavior***. One reason could be the use of several attributes in customizing the app navigation.

Up Navigation: All screens in an app, except the home screen, should offer users a way to navigate to the logical parent screen in the app's hierarchy by pressing the Up button in the action bar. It is implemented by declaring the logical parent activity in each activity using parentActivityName attribute of the <activity> element. The attribute was introduced in API level 16. For the apps that support API level 15 and lower, developers should include a <meta-data> inside the <activity> then specify the parent activity as the value of android.support.PARENT_ACTIVITY.

In the dataset, a parent activity has been added, removed, or changed in 168 apps. Developers have added a parent activity in 259 activities of 110 apps, including support.PARENT_ACTIVITY in 41 activities of 20 apps. We observed that developers have initiated the changes in most of the cases. However, in some cases, the changes were influenced by bug reports, user requests, or lint warnings. Developers have also changed parent activity in 121 activities of 42 apps. We observed that the changes were mostly caused by addition, removal, or renaming of the activities. In some cases, developers forgot to change the parent activity after the activity was removed, resulting in ActivityNotFoundException. Some developers have complained about using shorthand in the parent activity name in the <meta-data>. Therefore, they have used the fully qualified name. Finally, developers have removed parent activity from 78 activities of 44 apps. In most of the cases, it was affecting the normal flow of activities. In some cases, it was creating a problem with the activities task stack and their states because the logical parent activity was already available in the task stack. ***Although developers should offer parent activity in each activity, they should check its effect on activities task stack and their states***.

5.6 Device orientation

Android devices support various screen orientations such as portrait and landscape. Developers should provide support for all the orientations in their apps for maximum user reachability. However, app screens can be fixed to be viewed in a specific orientation by using screenOrientation attribute of <activity> elements, which is not a good practice unless it is required to support app features such as playing games in landscape mode.

In the dataset, screen orientation has been fixed, removed, or changed in 159 apps. Developers have fixed orientation of 234 screens (activities) in 109 apps, including 208 screens that have been fixed to portrait orientation. Only 26 screens have been fixed to landscape, sensorLandscape, sensorPortrait, nosensor, fullSensor, locked, or user orientation. ***We observed two main reasons for fixing screen orientation. First, device rotation caused errors in apps. Second, the inconvenience of using apps in other orientations, which also means developers may not have explicitly designed layout for other orientations.*** In several cases, developers have explicitly mentioned that the change is a temporary solution, which will be fixed later. Indeed, developers have removed the screen orientation restriction from 218 screens of 82 apps. However, it includes only 44 screens that have been fixed. The main reason is obviously to support different devices with various device orientations. However, we also found some cases where the restriction has been removed to support the multi-window display, which has been introduced in API level 24. Finally, developers have changed screen orientation in 47 screens of 21 apps, mostly to support specific device features available in a specific orientation. For example, landscape and portrait orientations have been changed to sensorLandscape and sensorPortrait, respectively in 32 screens.

When a device is rotated, Android restarts the running activity to reflect the new orientation. Developers should properly handle the activity lifecycle during restart so that state and data are not lost (Jha et al., 2019, Amalfitano et al., 2018, Shan et al., 2016, Adamsen et al., 2015, Zaeem et al., 2014). Developers can also prevent the activity from the restart by declaring a configChanges attribute with the value "orientation" in the <activity> element. Moreover, they should also declare "screenSize" in the attribute if the app targets API level 13 or higher. However, preventing the activity from restart during orientation change is not recommended (Runtime configuration change, 2018) because it can affect other elements of an app, which may lead to runtime errors.

In the dataset, configChanges attributes with a value orientation, screenSize, or both have been added or removed from 174 apps. The attribute has been added to 443 activities of 162 apps. ***We observed that, in most of the cases, screen orientation change resulted in state loss, including the destruction of running dialog windows. It also resulted in crashes in several apps.*** Therefore, developers prevented the activities from the restart by setting the attribute. In contrast to preventing the restart, developers have removed the attribute from 117 activities of 43 apps, including 45 activities in which the attribute was added. In some cases, developers have explicitly noted that setting the attribute is certain to cause issues. However, ***we hardly observed any cases where the configChanges attribute was removed because it actually resulted in errors***. In most of the cases, the attribute was removed after fixing the state loss issues.

5.7 Package name

Each app must have a package name declared in the Android manifest file. The package name is generally used as an app ID that must be unique in the Play store. However, the package name and the app ID can be different in an app. Once the app is published in the Play store, the new version of the app must have the same app ID; otherwise, the new version of the app is considered as a different app.

In the dataset, developers have changed the package name of 178 apps. In most of the cases, the package name has been changed at the very early stage of the app development. For example, the package name of 91 apps has been changed within the first three commits. Obviously, the change is mainly influenced by the change in the underlying package structure of the apps. However, after careful observation of the manifest file and the source code of the components, we found that several apps in the dataset are cloned from existing apps. Therefore, developers have changed the package name of the apps.

5.8 App icon

An app icon is declared using an icon attribute in the <application> element of the Android manifest file. It represents an icon for the app and the default icon for each of the app's components. An app icon is one of the critical factors that influence users' choice of apps (Lim et al., 2015).

In our dataset, the value of the icon attribute has been changed 315 times in 237 apps. One of the main reason for the change is moving app icons in a different resource folder. Developers have moved app icons from drawable to mipmap resource folder in 118 apps for better scaling. Android added support for mipmap starting from API level 18. We also found that developers have moved app icons from mipmap to drawable resource folder in 32 apps. However, they reverted the change in 16 apps. There are some other reasons for the change in the value of the icon attribute. In several cases, developers have changed the name of app icons to follow the common naming conventions (App icon naming, 2018) and resolve conflicts with the name of other resource files. We found very few cases where the actual design of the app icons has changed. Overall, ***developers have changed app icons to improve the quality of the icons and remove errors.***

Starting from API level 25, Android provides support for a round icon, which is declared using a roundIcon attribute in the <application> element. In the dataset, round icons have been added, changed, and removed from 45, 12, and 8 apps, respectively.

5.9 App theme

A theme is a type of style that is applied to an entire app or an activity. It is implemented by declaring a theme attribute in the <application> or <activity> elements of the Android manifest file. If different themes are declared in the <application> and the <activity> elements, the activity theme overrides the app theme. The Android platform provides various built-in themes and support libraries for themes. Developers can also use their custom themes.

In the dataset, themes have been added, removed, or changed in 401 apps, which means ***a large number of apps (56%) either do not use a theme or do not change the initial theme***. We observed several interesting patterns behind the change. In most of the apps that do not have any themes, developers have added material design theme provided by the support library to improve the user interface. In very few apps, developers have added custom themes. Developers have changed themes for various reasons, mainly to add or remove action bar, toolbar, and dialog, change support library, migrate from custom them to material design theme, switch between light and dark theme, change the background color, and fix UI bugs. Developers have removed themes mainly to fix UI bugs caused by themes or underlying support library. In several apps, themes have been removed from the <application> element and implemented in <activity> elements for flexibility.

5.10 Backup and restore

Android allows apps to backup and restore the user's data. Starting from API level 23, Android performs auto backup. The feature is implemented by using allowBackup attribute in the <application> element to enable or disable backup and setting a fullBackupContent attribute in the <application> element to include or exclude certain files for backup. The default value of allowBackup is true. Therefore, apps that target API level 23 or higher automatically performs the backup.

In our dataset, allowBackup="true" has been explicitly added in 66 apps and removed from 8 apps. These actions do not affect the end result because the value of allowBackup is true by default. However, the Android lint provides a warning if the value is not set explicitly. We also found that the value of the attribute has been changed to true in 5 apps. In contrast, the value of the attribute has been changed to false in 39 apps. Moreover, allowBackup="false" has been explicitly added in 10 apps. Overall, the auto backup was disabled in 49 apps. When the auto backup feature is enabled, the system backs up almost all app data by default. However, developers can include or exclude certain files by including a fullBackupContent attribute with a value that points to an XML file. In the dataset, developers have added the fullBackupContent attribute in 85 apps. However, they have included a Boolean value in 52 apps, which is incorrect. The apps with the incorrect attribute values will perform the default behavior, which may have some privacy issues (Panda, 2016, XDA Developers, 2015). Developers have later corrected the error in 6 apps by removing the attributes with incorrect values or changing the value to an XML file.

6. Discussion

In this section, we discuss the empirical results in the context of three different research questions proposed in Section 4.3. Furthermore, we discuss the research implications for practitioners and researchers.

6.1 RQ1: Most frequently changed attributes

As shown in Fig. 4, app components are the most frequently changed attributes in Android apps. App components have been added or removed from 696 apps. Out of total 12,341 commits analyzed, 4,459 commits (36%) have been used partially or completely to add or remove components. The result is not surprising because the core functionalities of Android apps are implemented through the components. As observed by Lim et al. (2015), user needs and trends change quickly for mobile apps. Clearly, most of the apps in the dataset change functionalities over time by extending the current functionalities or discarding unused functionalities. However, the result shows that ***functionalities addition or removal over time does not depend on the number of functionalities offered by the apps***. Moreover, in comparison to discarding functionalities by removing components, most of the apps extend current functionalities over time by adding components. The result also shows that, in addition to specific functionalities related to a particular app, developers have added general functionalities in several apps. Therefore, ***developers should include general screens such as about and setting screens in the apps for consistency with user navigation behavior***.

We can also observe from Fig.4 that a significant number of apps (23%) do not change functionalities over time. Moreover, as discussed in Section 5.1, ***the apps that do not change their functionalities over time are not negatively affected in their success***. To some extent, the result contradicts the observation made by Lim et al. (2015). They observed that 38 percent of users abandon apps because they are bored of them and 44 percent of users abandon apps because they are no longer needed. However, it does not mean that the users who abandoned the apps gave negative reviews. We are not claiming that the apps that do not change their functionalities over time are positively affected in their success. As also observed by Nayebi et al. (2018), mobile apps appear to challenge Lehman's sixth law of software evolution (Lehman, 1996), which states that "functional content of a program must be continually increased to maintain user satisfaction over its lifetime".

In the top-ten list of the most frequently changed attributes, there are two attributes that are related to app components: components' name and components' directory. Components' name has been changed in 228 apps, mainly to reflect the component type and its functionality; whereas, components' directory has been changed in 248 apps, mainly to manage app components according to their type. Standard naming conventions play an important role in code comprehension (Naming Conventions, 2018) and maintenance. Therefore, ***developers should always use a descriptive component name that reflects the core functionality of the component and its type. Moreover, they should always create separate directories for each component type and place the files representing the components in separate directories, which will certainly improve the maintainability of the apps***.

The second in the list of top-ten most frequently changed attribute is app permission, which has been added or removed from 527 apps. There are two main reasons for the change in app permission usage. The first reason is directly related to change in app functionalities. The empirical result in Section 5.1 clearly shows that a large number of apps add or extend functionalities; whereas, comparatively a small number of apps remove functionalities. We can observe the same pattern with app permission usage in the empirical results in Section 5.4. Therefore, permission usage is directly related to app functionalities. For example, we observed in several apps that a boot receiver component and RECEIVE_BOOT_COMPLETED permission are added together in the same commit. The second reason for the change in permission usage over time is platform-induced changes. The system permission has gone through several changes over time, including a major overhaul in API level 23. Several system permissions have been deprecated and new permissions have been introduced, which has contributed to the result of app permission as one of the most frequently changed attributes in Android apps. ***We have clearly observed in this study that developers have become more sensitive towards permission request, which is a significant positive deviation from earlier practices of permission usage***.

In addition to security, privacy is one of the major concerns among app users and developers. One of the attributes in the list of top-ten most frequently changed attributes is auto backup, which can create privacy issues (Panda, 2016, XDA Developers, 2015). We observed that most of the apps enable the auto backup feature explicitly. Meanwhile, they restrict data during auto backup. Moreover, a significant number of apps disable auto backup. The change is mostly influenced by Android lint warnings. However, we also observed that developers are incorrectly implementing data restriction during auto backup. Therefore, ***similar to a lint warning for explicitly setting allowBackup value, platform developers should provide a lint warning for incorrect values in fullBackupContent attributes***.

The user interface is one of the critical parts of Android apps. ***There is a constant urge among app developers to improve the user interface, which can be observed from the fact that app theme, navigation, and icon are in the list of top-ten most frequently changed attributes in Android apps***. For example, most of the apps have added theme and navigation to improve the user interface. Another major reason is the platform-induced change. Unlike platform-induced permission change where developers must implement the change in apps, platform-induced changes in the user interface are mostly optional for developers. However, developers have adopted the changes to improve the overall quality of apps. For example, developers have changed the app icon resource folder from drawable to mipmap to improve the quality of icons. Similarly, developers have added up navigation and implemented material design theme to improve the overall user experience. However, we also observed that developers are facing difficulties in customizing app navigation due to a large number of attributes affecting the navigation behavior. The default navigation behavior does not satisfy navigation requirements for several apps. Therefore, ***platform developers should provide several default navigation patterns instead of a single default navigation behavior***.

To our surprise, the attribute package name is in the list of top-ten most frequently changed attribute. A package name generally represents a unique app ID. Therefore, the change in package name implies that the app is a new app, at least in the Google Play. However, in most of the apps, we observed that the package name has been changed at the very early stage of app development, probably before publishing in the Play store. Moreover, we observed that several apps have been cloned from existing apps, resulting in a change in the package name.

Bugs cannot be ignored in this discussion, which is one of the major influencing factors behind overall configuration changes. Specifically, the app state and UI bugs (Jha et al., 2019) are the major reasons behind the change in device orientation. As discussed in Section 5.6, developers have either fixed the orientation or prevented configuration changes during orientation in several apps to prevent the bugs. The app state and UI bugs have also influenced changes in several other attributes such as app navigation and theme. Therefore, ***developers should use default navigation behavior unless the app has a very specific requirement. Moreover, they should create different screen layouts for different screen orientations or fix the screens to a specific orientation***. Furthermore, developers can use several tools and techniques (Farooq and Zhao, 2018, Kowalczyk et al., 2018, Shan et al., 2016, Adamsen et al., 2015, Zaeem et al., 2014) to prevent bugs caused by runtime configuration changes.

HIGHLIGHTS

- Most of the apps extend current functionalities, including sensitive functionalities, by adding components and permissions. However, functionalities addition or removal over time does not depend on the number of functionalities offered by the apps. Moreover, the apps that do not change their functionalities over time are not negatively affected in their success.
- Most of the apps improve the design and functionalities of the user interface by changing attributes such as theme, icon, and navigation. App developers are constantly making unsolicited changes in the user interface to improve overall app quality.
- Platform-induced changes, particularly related to security and privacy, are also major influencing factors behind configuration change.

6.2 RQ2: Adoption pattern of new attributes

The goal of this research question is to understand the adoption pattern of different attributes and elements introduced in different versions of the Android platform. To get the list of attributes and elements introduced in different versions of the Android platform, we manually went through the Android documentation of API change history (SDK Platform release notes, 2018) and the Android manifest file (App Manifest Elements, 2018). The first column of Table 5 shows the 23 attributes and 2 elements introduced by the platform since API level 8. The second column shows the API level in which the element or attribute was introduced. The third column shows the numbers of apps in the dataset that have been created before the element or attribute was introduced. For example, the dataset has 23 apps that were created before the attribute installLocation was introduced in API level 8 (May 2010). The last column represents the percentage of existing apps adopting the introduced element or attribute. For example, 26.08% of apps (6 apps out of 23 apps) adopted the installLocation attribute.

Table 5. Elements and attributes introduced by the Android since API level 8.

Attributes/Elements	**API Level**	**Total Apps**	**Adoption (%)**
installLocation	8	23	26.08
backupAgent	8	23	4.35
restoreAnyVersion	8	23	4.35
vmSafeMode	8	23	0.00
compatible-screens	9	35	0.00
xlargeScreens	9	35	11.43
requiresSmallestWidthDp	13	52	0.00
compatibleWidthLimitDp	13	52	0.00
largestWidthLimitDp	13	52	0.00
uiOptions	14	60	1.67
parentActivityName	16	118	6.78
supportsRtl	17	150	7.33
restrictedAccountType	18	205	0.00
requiredAccountType	18	205	0.00
maxSdkVersion	19	235	2.55
maxRecents	21	352	0.00
persistableMode	21	352	0.00
usesCleartextTraffic	23	531	0.00
uses-permission-sdk-23	23	531	0.38
supportsPictureInPicture	24	743	0.27
networkSecurityConfig	24	743	0.81
resizeableActivity	24	743	3.90
roundIcon	25	771	4.80
appCategory	26	895	0.34
targetProcesses	26	895	0.00

As shown in the table, out of 25 elements and attributes introduced by the platform since API level 8, 11 elements and attributes have not been adopted by any existing apps. Furthermore, out of the remaining 14 elements and attributes, 10 attributes and elements are adopted by less than 5% of the existing apps. The result does not include the attributes or elements that have been adopted incorrectly. ***The result clearly shows that the percentage of existing apps adopting the newly introduced attributes or elements is very low. Android platform developers need to either reconsider the usefulness of the introduced attributes or find different ways to communicate with the app developers about the usefulness of the introduced attributes***. A qualitative study by Sayagh et al. (2018) shows the need for actively maintaining dead and unused configuration options.

We also investigated the lag time in adopting newly introduced attributes and elements. The result is shown in Fig. 8. We can observe from the figure that most of the apps (68%) adopted the introduced attributes and elements within the first 12 months. However, if we separate the lag time based on the API level, we can clearly observe that the lag time for the attributes or elements introduced before API level 23 is 13 to 57 months in most of the apps (77%); whereas, the lag time for the attributes and elements introduced since API level 23 is within 12 months in most of the apps (94%). Therefore, ***the adoption lag time for the newly introduced attributes and elements is decreasing over time***. There could be several reasons behind the decreasing lag time. However, the main reason could be the release of the developers preview version before the official public release since API level 23. Another major reason could be the improvement in app development tools such as Android lint.

HIGHLIGHTS

- The percentage of existing apps adopting the newly introduced attributes or elements is very low.
- The adoption lag time for the newly introduced attributes and elements is significantly high. However, it is decreasing over time.

6.3 RQ3: Effort distribution pattern

The main goal of this research question is to identify wasted effort by understanding developers' effort distribution in configuration changes. The distribution of developers' effort at commit-level is shown in Fig. 9. To identify one-time commits and the commits that are further changed or reverted, we created linked chains of commits in an app. We explain the process through an example. Suppose a developer added a service component and a theme in an activity component in the first commit of an app. In the second commit, the developer added an app icon. In our analysis, we identify the first commit and the second commit as C1 and C2, respectively. Let us assume that the developer made a new commit in which she changed the theme of the activity component, which was added in commit C1. The new commit will be identified as PCC1 (Partially Changed Commit 1) because commit C1 was partially changed. Let us again assume that the developer made another commit in which app icon was removed. The new commit will be identified as RC2 (Reverted Commit 2). In this manner, the process creates linked chains of commits, which were used to identify one-time commits and the commits that were further changed or reverted.

Out of 12,341 commits made by developers, 5,865 commits (47%) are permanent or one-time commits, which means configurations in these commits are not changed again. However, 2,856 commits (23%) are further changed or reverted partially or completely using the remaining 3,620 commits (30%). Specifically, 2,040 commits (17%) are partially or completely used to revert already changed configuration; whereas 1,580 commits (13%) are partially or completely used to further change the already changed configuration. The result clearly shows that the effort is equally distributed between a large number of one-time configuration change (47% of commits) and a small number of frequently changed configuration (23% of commits). The frequent change in the configuration does not always result in wasted effort. However, the reversal of changes mostly results in wasted effort.

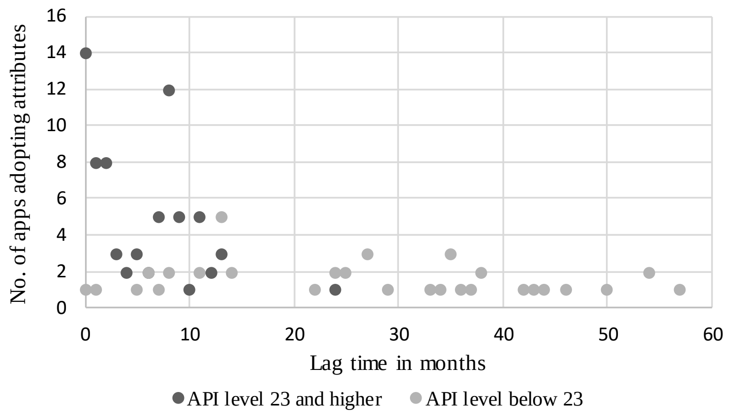


Fig. 8. Time-lag in adopting new attributes and elements.

The result shows that significant effort is wasted in changing configuration and then reverting back the change. For example, the result in Section 5.1 shows that more than 50% of the removed components are first added then removed. Moreover, the time interval between addition and removal of the components is very short in most of the cases, which suggests that the changes are practically ineffective, resulting in wasted effort. One may think that changing attributes in the Android manifest file does not require much effort. However, in several cases, changing an attribute in the Android manifest file requires a series of changes in the underlying source code. For example, to add an activity component in the Android manifest file, a developer has to write an entire Java class and an XML layout file. Furthermore, if the Java class uses sensitive resources, the developer has to declare permissions in the Android manifest file. Moreover, if the app targets API level 23 and more, the developer has to implement runtime permission in the source code. Therefore, adding a single component in the Android manifest file requires significant effort, which is wasted if the component is removed.

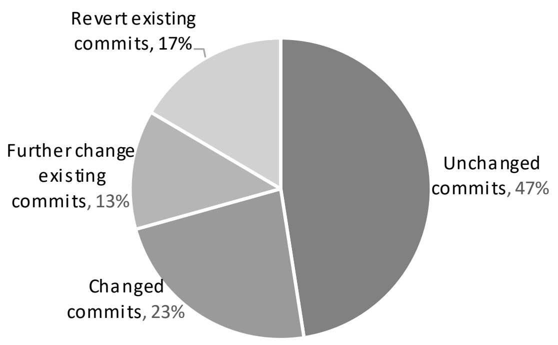


Fig. 9. Effort distribution in configuration changes.

We found that more than 48% of the reverted configuration changes cover addition and removal of app components and permissions, which are also the two most frequently changed attributes. Therefore, developers should carefully think about the app requirements before extending current functionalities or adding new functionalities in the app (Guerrouj et al., 2015). We recommend that ***developers should communicate with the app user base prior to changing their app features***. For example, several researchers have proposed tools and techniques to extract requirements from user feedbacks (Iacob and Harrison, 2013, Galvis-Carreno and Winbladh, 2013, Guzman and Maalej, 2014, Vu et al., 2015, Sorbo et al., 2016, Palomba et al., 2017, Jha and Mahmoud, 2017) and prioritize them (Chen et al., 2014, Gao et al., 2015, Keertipati et al., 2016, Scalabrino et al., 2017). Furthermore, developers should take some proactive measures to reduce the effort in app maintenance. For example, they should use descriptive component names that reflect the core functionality of the components and their types, they should create different directories for each component type, they should use default navigation behavior unless the app has a very specific requirement, they should create different screen layouts for different screen orientations or fix the screens to a specific orientation, and they should implement theme in an individual screen rather than in the whole app.

HIGHLIGHTS

- The effort is equally distributed between a large number of one-time configuration change (47% of commits) and a small number of frequently changed configuration (23% of commits).
- Significant effort is wasted in changing configuration and then reverting back the change.

6.4 Implications for practitioners and researchers

The results presented in this paper can be valuable for both practitioners (app and platform developers) and researchers in improving the app development and maintenance activities.

App developers can take several proactive measures that can help them in reducing effort and improving the quality and maintainability of their apps. The result of this paper shows that most of the app maintenance activities are concentrated on features addition or removal. Although most of the apps added features, several of them removed the added features. We also found that the apps that do not change their features over time are not negatively affected in their success. Therefore, developers should communicate with their user base prior to changing their app features. Moreover, as suggested by Sarro et al. (2018), they should check the features of the popular apps in the same app category to gain information about the successful features. However, they should add general features such as setting and about screens for consistency with the user navigation behavior. We also found that features addition or removal over time does not depend on the number of features offered by the apps. Therefore, developers should not think that a large number of features in an app will definitely create maintainability issues. However, some of the attributes that can create maintainability issues are unintuitive component names and unstructured component directories. Therefore, developers should always use descriptive component names that reflect the core functionality of the components and their types and they should create different directories for each component type. The result of this paper shows that user interface is another area in which app maintenance activities are mostly concentrated. We found that developers have added app icons and themes to improve the overall quality of the apps. Therefore, developers should always include app icons and themes in their apps. Moreover, consistency is a key factor when designing the user interface.

It should be consistent with user interaction behavior; otherwise, users may abandon the apps. Therefore, developers should always use default navigation behavior and default themes, unless they have very specific requirements. We found that customizing default navigation behavior or default themes does not only create inconsistencies in user interaction behavior but also may cause severe bugs. The result of this paper also shows that developers make a significant effort in updating security and privacy issues. Although these updates are mostly caused by platform-induced changes, similar to other platform-induced changes, developers often do not make these updates in time and they make errors in implementation. Therefore, developers should check the platform release notes (SDK Platform release notes, 2018) for any updates.

Platform developers can get valuable information from this study to improve the platform for app developers. We found in this study that app developers are making errors in implementing the newly introduced attributes. For example, they are placing incorrect values in the fullBackupContent attribute to implement the backup and restore feature. Although the capability of the Android lint in detecting errors has improved significantly since we performed the study on developer mistakes in writing the Android manifest file (Jha et al., 2017)), there are still scopes for improvement. Moreover, critical issues such as security and privacy should not be ignored by the Android lint. In this study, we also found that app developers are facing difficulties in customizing the default behavior, especially for app navigation. Moreover, the default behavior is not sufficient for many apps. Therefore, platform developers should provide several default navigation patterns instead of a single default navigation behavior. It will certainly ease the implementation task for app developers. Finally, we found that several of the newly introduced attributes are either not adopted or adopted scarcely. Moreover, the adoption lag time is still high for such a rapidly evolving platform. Therefore, platform developers should perform some kind of surveys among app developers before introducing any attributes to understand their usefulness. Moreover, they should periodically check the adoption of newly introduced attributes and remove the attributes that have not been adopted for a long time. Furthermore, similar to developers preview release, platform developers should find various ways to reach app developers to promote the newly introduced attributes.

Other than app reviews analysis and API evolution, there are hardly any research works that focus on app maintenance and evolution. Therefore, the results of this study can become the basis for new research directions in the area of app maintenance and evolution. For example, researchers can develop tools and techniques to predict features and other attributes addition or removal based on the commit history analysis of the same category apps. Similarly, researchers can also develop tools and techniques to predict the effort required to maintain apps. These predictions can help app developers in improving app quality and reducing app maintenance effort.

7. Threats to validity

Threats to external validity relate to the generalizability of our results. In comparison to millions of apps available for download in the Google Play store, we performed the empirical study on a very small dataset with 908 free and open source apps. Therefore, the results of this study may not be generalized in a larger context. However, to the best of our knowledge, this is the first study on configuration changes in Android apps. Moreover, our dataset is larger than the existing studies (Wei et al., 2012, Calciati and Gorla, 2017) on permission evolution. Furthermore, we performed the study on only free and open source Android apps. Therefore, the results may not be generalized for commercial apps.

Threats to internal validity relate mainly to the methodology used to perform this study. We collected commit history of the Android manifest files from GitHub repositories. There may be the cases where commit history data were lost during project import in GitHub from other repositories. Moreover, manifest files may have changed prior to placing the app in a versioning system (Minelli and Lanza, 2013). The commit data may also have been lost due to developers' mistakes. This study does not account for the lost historical data. We extracted configuration changes data manually by reading each commit history. Although GitHub highlights the changed information, we may have made errors in extracting the data. To mitigate human errors, the first and the second author verified the extracted data by cross-checking. We extracted attributes and elements introduced by the Android platform since API level 8 by going through the Android documentation. We may have missed some attributes or elements due to incomplete documentation. In Section 5.1, we have identified the component removal and then addition by using the component class name. However, it may be the case where the component has been removed and then added with a different class name. In Section 6.3, we have identified the wasted effort as a reversal of configuration changes such as addition and then removal of functionalities. However, it may be the case where more effort would require to change a functionality than removing the functionality and adding a new functionality (Nayebi et al., 2018). Our result is still valid because the effort is wasted in adding and then removing the functionality.

8. Conclusion

In this paper, we performed an empirical study to understand app evolution through configuration changes by analyzing 12,341 commit histories of Android manifest files of 908 free and open source Android apps. The result clearly shows that most of the apps extend current functionalities, including sensitive functionalities, by adding components and permissions. However, the apps that do not add or remove functionalities over time are not negatively affected in their success. Moreover, the changes in app functionalities do not depend on the number of functionalities offered by the apps. The result also shows that most of the apps improve the design and functionalities of the user interface by changing attributes such as theme, icon, and navigation. While platform-induced changes and bugs are two major influencing factors behind configuration changes, it is mainly influenced by the changes in app functionalities. Obviously, the effort is mostly concentrated on changing app functionalities and improving the user interface. However, significant effort is wasted in changing app functionalities and then reverting back the change. The result also indicates that a very small percentage of existing apps adopt new attributes or elements introduced by the platform. Furthermore, the lag time in adoption is mostly within 12 months. However, it is decreasing over time. Overall, the results in this paper can help both app and platform developers in the maintenance and evolution of Android apps and the platform, respectively.

Acknowledgment

This research was supported by the BK21 Plus project (SW Human Resource Development Program for Supporting Smart Life – 21A20131600005), Kyungpook National University, Korea and the Basic Science Research Program through the National Research Foundation of Korea (NRF-2017R1D1A3B04035880 and NRF-2018R1A6A1A03025109) funded by the Ministry of Education.

References

Adamsen CQ, Mezzetti G, Møller A (2015) Systematic execution of android test suites in adverse conditions. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp 83–93. ACM.

Alharbi K, Yeh T (2015) Collect, decompile, extract, stats, and diff: Mining design pattern changes in Android apps. In Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services, pp. 515-524. ACM.

Amalfitano D, Riccio V, Paiva AC, Fasolino AR (2018) Why does the orientation change mess up my Android application? From GUI failures to code faults. Software Testing, Verification and Reliability, 28(1), e1654.

AppBrain (2018) Number of available Android apps in the Play Store. http://www.appbrain.com/stats/number-of-android-apps.

App icon naming (2018) App icon naming convention: https://developer.android.com/guide/practices/ui_guidelines/icon_design.html#design-tips.

App Manifest Overview (2018) App Manifest Overview - https://developer.android.com/guide/topics/manifest/manifest-intro.

App Manifest Elements (2018) Android Manifest Elements - https://developer.android.com/guide/topics/manifest/manifest-element.

Barrera D, Kayacik HG, Oorschot PCV, Somayaji A (2010) A methodology for empirical analysis of permission-based security models and its application to android. In Proceeding of the 17th ACM conference on Computer and communications security, pp. 73-84. ACM.

Best practices (2018) Android developers' best practices - https://developer.android.com/distribute/best-practices/.

Beyer S, Pinzger M (2014) A manual categorization of android app development issues on stack overflow. In 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 531-535. IEEE.

Calciati P, Gorla A (2017) How do apps evolve in their permission requests?: a preliminary study. In 14th International Conference on Mining Software Repositories, pp. 37-41. IEEE Press.

Calciati P, Kuznetsov K, Bai X, Gorla A (2018) What did Really Change with the new Release of the App? In 15th International Conference on Mining Software Repositories (MSR).

Carreño LVG, 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.

Chang N, Wang L, Pei Y, Mondal SK, Li X (2018) Change-Based Test Script Maintenance for Android Apps. In 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 215-225. IEEE.

Chen N, Lin J, Hoi SC, 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.

Dataset (2018) Dataset of configuration change in Android apps: https://github.com/HiFromAjay/ConfigChange/blob/master/Dataset.pdf.

Dehlinger J, Dixon J (2011) Mobile application software engineering: Challenges and research directions. In Workshop on mobile software engineering (Vol. 2, pp. 29-32).

Do Q, Yang G, Che M, Hui D, Ridgeway J (2016) Regression test selection for android applications. In 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 27-28. IEEE.

Enck W, Octeau D, McDaniel PD, Chaudhuri S (2011) A study of android application security. In USENIX security symposium (Vol. 2, p. 2).

F-Droid (2018) Free and Open Source Android App Repository: https://f-droid.org/en/.

Fan L, Su T, Chen S, Meng G, Liu Y, Xu L, Pu G, Su Z (2018). Large-Scale Analysis of Framework-Specific Exceptions in Android Apps. In ICSE '18: 40th International Conference on Software Engineering

Farooq U, Zhao Z (2018) RuntimeDroid: Restarting-Free Runtime Change Handling for Android Apps. In 16th Annual International Conference on Mobile Systems, Applications, and Services. ACM

Flora HK, Chande SV, Wang X (2014a) Adopting an agile approach for the development of mobile applications. International Journal of Computer Applications, 94(17).

Flora HK, Wang X, Chande SV (2014b) An investigation into mobile application development processes: Challenges and best practices. International Journal of Modern Education and Computer Science, 6(6), 1.

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 LV, 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.

Gao C, Wang B, He P, Zhu J, Zhou Y, Lyu MR (2015) Paid: Prioritizing app issues for developers by tracking user reviews over versions. In 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp. 35-45. IEEE.

Genc-Nayebi N, Abran A (2017) A systematic literature review: Opinion mining studies from mobile app store user reviews. Journal of Systems and Software, 125, 207-219.

Guerrouj L, Azad S, Rigby PC (2015) The influence of app churn on app success and stackoverflow discussions. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 321-330. IEEE.

Guzman E, Maalej W (2014) How do users like this feature? a fine grained sentiment analysis of app reviews. In 2014 IEEE 22nd International Requirements Engineering Conference (RE), pp. 153-162. IEEE.

He D, Li L, Wang L, Zheng H, Li G, Xue J (2018). Understanding and detecting evolution-induced compatibility issues in Android apps. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 167-177. ACM.

Hecht G, Benomar O, Rouvoy R, Moha N, Duchien L (2015) Tracking the software quality of android applications along their evolution (t). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 236-247. IEEE.

Iacob C, Harrison R (2013) Retrieving and analyzing mobile apps feature requests from online reviews. In Proceedings of the 10th Working Conference on Mining Software Repositories (pp. 41-44). IEEE Press.

Interaction model (2018) Activity launch mode: https://developer.android.com/guide/topics/manifest/activity-element#lmode.

Inukollu VN, Keshamoni DD, Kang T, Inukollu M (2014) Factors influencing quality of mobile apps: Role of mobile app development life cycle. arXiv preprint arXiv:1410.4537.

Jabangwe R., Edison H, Duc AN (2018) Software engineering process models for mobile app development: A systematic literature review. Journal of Systems and Software, 145, 98-111.

Jha AK, Lee S, Lee WJ (2019) Characterizing Android-specific crash bugs. In 6th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft). 2019, IEEE.

Jha AK, Lee S, Lee WJ (2017) Developer mistakes in writing Android manifests: an empirical study of configuration errors. In 14th International Conference on Mining Software Repositories (MSR), pp. 25-36. 2017, IEEE.

Jha AK, Lee WJ (2018) An empirical study of collaborative model and its security risk in Android. Journal of Systems and Software (JSS). 137:550-562.

Jha AK, Lee WJ (2016) Analysis of Permission-based Security in Android through Policy Expert, Developer, and End User Perspectives. J. UCS, 22(4), 459-474.

Jha N, Mahmoud A (2017) Mining user requirements from application store reviews using frame semantics. In International Working Conference on Requirements Engineering: Foundation for Software Quality, pp. 273-287. Springer.

Joorabchi ME, Mesbah A, Kruchten P (2013) Real challenges in mobile app development. In 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 15-24. IEEE.

Keertipati S, Savarimuthu BTR, Licorish SA (2016) Approaches for prioritizing feature improvements extracted from app reviews. In Proceedings of the 20th international conference on evaluation and assessment in software engineering, p. 33. ACM.

Kim JM, Porter A (2002) A history-based test prioritization technique for regression testing in resource constrained environments. In Proceedings of the 24th international conference on software engineering, pp. 119-129. ACM.

Kowalczyk E, Cohen MB, Memon AM (2018) Configurations in Android testing: they matter. In Proceedings of the 1st International Workshop on Advances in Mobile App Analysis (pp. 1-6). ACM.

Lehman MM (1996) Laws of software evolution revisited. In European Workshop on Software Process Technology, pp. 108-124. Springer, Berlin, Heidelberg.

Li L, Bissyandé TF, Le Traon Y, Klein J (2016) Accessing inaccessible android apis: An empirical study. In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 411-422. IEEE.

Li L, Gao J, Bissyandé TF, Ma L, Xia X, Klein J (2018a) Characterising deprecated android apis. In Proceedings of the 15th International Conference on Mining Software Repositories, pp. 254-264. ACM.

Li L, Bissyandé TF, Wang H, Klein J (2018b) Cid: automating the detection of api-related compatibility issues in android apps. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 153-163. ACM.

Li X, Chang N, Wang Y, Huang H, Pei Y, Wang L, Li X (2017) ATOM: Automatic Maintenance of GUI Test Scripts for Evolving Mobile Applications. In 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 161-171. IEEE.

Li Z, Harman M, Hierons RM (2007) Search algorithms for regression test case prioritization. IEEE Transactions on software engineering, 33(4).

Lim SL, Bentley PJ (2013) Investigating app store ranking algorithms using a simulation of mobile app ecosystems. In: 2013 IEEE Congress on Evolutionary Computation (CEC), pp 2672–2679

Lim SL, Bentley P, Kanakam N, Ishikawa F, Honiden S (2015) Investigating country differences in mobile app user behavior and challenges for software engineering. IEEE Transactions on Software Engineering, (1), 1-1.

Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Penta MD, Oliveto R, Poshyvanyk D (2013) API change and fault proneness: a threat to the success of Android apps. In Proceedings of the 2013 9th joint meeting on foundations of software engineering, pp. 477-487. ACM.

Linares-Vásquez M, Dit B, Poshyvanyk D (2013a) An exploratory analysis of mobile development issues using stack overflow. In 2013 10th IEEE Working Conference on Mining Software Repositories (MSR), pp. 93-96. IEEE.

Lynch J (2012) App store optimization: 8 tips for higher rankings. https://searchenginewatch.com/sew/how-to/2214857/app-store-optimization-8-tips-for-higher-rankings

Maalej W, Nabil H (2015) Bug report, feature request, or simply praise? on automatically classifying app reviews. In 2015 IEEE 23rd international requirements engineering conference (RE), pp. 116-125. IEEE.

Martin W, Sarro F, Harman M (2016) Causal impact analysis for app releases in google play. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 435-446. ACM.

McDonnell T, Ray B, Kim M (2013) An empirical study of api stability and adoption in the android ecosystem. In 29th IEEE International Conference on Software Maintenance (ICSM), pp. 70-79. IEEE.

McIlroy S, Ali N, Hassan AE (2016) Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. Empirical Software Engineering, 21(3), 1346-1370.

Miles MB, Huberman AM, Saldaña J (2013) Qualitative Data Analysis: A Methods Sourcebook (3rd ed.). SAGE Publications, Inc.

Minelli R, Lanza M (2013) Software Analytics for Mobile Applications--Insights & Lessons Learned. In 2013 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 144-153. IEEE.

Nagappan M, Shihab E (2016) Future Trends in Software Engineering Research for Mobile Apps. In FOSE@ SANER, pp. 21-32.

Naming conventions (2018) Naming convention - https://www.oracle.com/technetwork/java/codeconventions-135099.html

Nayebi M, Kuznetsov K, Chen P, Zeller A, Ruhe G (2018) Anatomy of Functionality Deletion. In 15th International Conference on Mining Software Repositories (MSR).

Palomba F, Linares-Vásquez M, Bavota G, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A (2018) Crowdsourcing user reviews to support the evolution of mobile apps. Journal of Systems and Software, 137, 143-162.

Palomba F, Salza P, Ciurumelea A, Panichella S, Gall H, Ferrucci F, Lucia AD (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.

Panda P (2016) What is Android allowBackup and How it Affects Your App - https://blog.devknox.io/what-is-android-allowbackup-how-it-affects-your-app/.

Panichella S, Sorbo AD, Guzman E, Visaggio CA, Canfora G, Gall HC (2015) How can i improve my app? classifying user reviews for software maintenance and evolution. In 2015 IEEE international conference on Software maintenance and evolution (ICSME), pp. 281-290. IEEE.

Panichella S, Sorbo AD, Guzman E, Visaggio CA, Canfora G, Gall HC (2016) Ardoc: App reviews development oriented classifier. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 1023-1027. ACM.

Permissions overview (2018) Permissions overview - https://developer.android.com/guide/topics/permissions/overview

Rakestraw TL, Eunni RV, Kasuganti RR (2013) The mobile apps industry: A case study. Journal of Business Cases and Applications, 9, 1.

Rosen C, Shihab E (2016) What are mobile developers asking about? a large scale study using stack overflow. Empirical Software Engineering, 21(3), 1192-1223.

Runtime configuration change (2018) Handle configuration changes: https://developer.android.com/guide/topics/resources/runtime-changes.

Saha RK, Zhang L, Khurshid S, Perry DE (2015) An information retrieval approach for regression test prioritization based on program changes. In Proceedings of the 37th International Conference on Software Engineering, pp. 268-279. IEEE Press.

Sarro F, Harman M, Jia Y, Zhang Y (2018) Customer rating reactions can be predicted purely using app features. In 2018 IEEE 26th International Requirements Engineering Conference (RE), pp. 76-87. IEEE.

Sayagh M, Kerzazi N, Adams B, Petrillo F (2018) Software Configuration Engineering in Practice: Interviews, Survey, and Systematic Literature Review. IEEE Transactions on Software Engineering.

Scalabrino S, Bavota G, Russo B, Oliveto R, Di Penta M (2017) Listening to the crowd for the release planning of mobile apps. IEEE Transactions on Software Engineering.

SDK Platform release notes (2018) SDK Platform release notes - https://developer.android.com/studio/releases/platforms

Seaman CB (1999) Qualitative methods in empirical studies of software engineering. IEEE Trans Softw Eng (IST) 25(4):557–572

Shan Z, Azim T, Neamtiu I (2016) Finding resume and restart errors in android applications. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 864-880. ACM.

Sorbo AD, Panichella S, Alexandru CV, Shimagaki J, Visaggio CA, Canfora G, Gall HC (2016) What would users change in my app? summarizing app reviews for recommending software changes. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 499-510. ACM.

Syer MD, Nagappan M, Hassan AE, Adams B (2013) Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source Android apps. In Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, pp. 283-297.

Taylor VF, Martinovic I (2017) To update or not to update: Insights from a two-year study of Android app evolution. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, pp. 45-57. ACM.

Tian Y, Nagappan M, Lo D, Hassan AE (2015) What are the characteristics of high-rated apps? a case study on free android applications. In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 301-310. IEEE.

Vu PM, Nguyen TT, Pham HV, Nguyen TT (2015). Mining user opinions in mobile app reviews: A keyword-based approach (t). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 749-759. IEEE.

Wasserman AI (2010) Software engineering issues for mobile application development. In Proceedings of the FSE/SDP workshop on Future of software engineering research, pp. 397-400. ACM.

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, ACSAC '12, pages 31–40.

XDA Developers (2015) Devs Beware – Automatic Backup Privacy Risks - https://www.xda-developers.com/devs-beware-automatic-backup-privacy-risks/.

Zaeem RN, Prasad MR, Khurshid S (2014) Automated generation of oracles for testing user-interaction features of mobile apps. In 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST), pp. 183-192. IEEE.