Developer mistakes in writing Android manifests: An empirical study of configuration errors

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

***Abstract*— Each Android app must have an Android manifest file. It is one of the most important configuration files manually written by developers. In addition to various configuration parameters required to run an app, it also contains configuration parameters which are used to implement security, compatibility, and accessibility of an app. Any mistakes in writing the manifest file can cause serious implications in terms of security, reliability, and availability of an app. In this paper, we study and report different types of mistakes committed by developers in writing Android manifest files. The study was performed on 13,483 real-world Android apps. We also present an open source rule-based static analysis tool which detects developer mistakes in the manifest file. The tool generates a warning message if it detects any misconfigurations in the manifest file. We used the tool to perform the empirical study and it generated total 59,547 configuration errors in 11,110 apps. Only 2,373 apps, among studied apps, do not have any configuration errors.**

***Keywords*- Android apps; Android manifest; configuration errors; rule-based error detection**

I. INTRODUCTION

Android operating system has become pervasive and ubiquitous. It can be found almost everywhere in diverse categories of devices for example, wrist watches, infotainment systems, tablets, and smartphones. With the penetration of Android operating system in diverse categories of devices, number of different purposes Android apps are constantly increasing in the market. As of February 2017, Google Play store has more than 2.6 million apps available for download [1]. Currently, the Play store hosts wide range of personal as well as business apps, including apps which perform critical tasks such as financial transactions, health monitoring, biometrics security, etc. Thus, there is a pressing demand from users for highly reliable and secure Android apps which is evident from the fact that users have discontinued using apps due to reliability and security issues [2].

Configuration errors are one of the leading cause of system failures [3] but, unlike many complex systems software where a configuration task is mainly performed by system administrators or users, an Android manifest file is written by developers. One can hypothesize that developers would make less mistakes because they have better understanding of the system as well as better programming and debugging skills. While it may be correct in comparison to system administrators or users, Android app developers encounter additional challenges in writing the manifest file. Most of the configuration parameters in the manifest file are highly specific to Android platform which even an experienced Java programmer must learn from scratch. An existing work [4] shows that Android specific bugs are more prevalent in Android apps than bugs related to an app logic. Another major challenge is security implementation. Security in Android [5, 6, 7] can be implemented at both system and application levels. At the application level, security is implemented largely through the manifest file so developers must have deep understanding of it but studies [8, 9] suggest quite opposite. In addition to Android specific challenges, generic challenges of writing a configuration file persist. One major generic challenge is dependency and correlation among configuration parameters.

It is irrefutable that developers make mistakes. Developers on Android platform are not exceptions. A quick search of Stack Overflow website through the keyword "*android manifest error*" retrieved 8,813 issues [10]. It suggests that developers are indeed making mistakes in writing the manifest file. One common mistake is using incorrect prefix when declaring permissions in the manifest. For example, *INSTALL_SHORTCUT* permission must use "*com.android.launcher.permission.*" prefix but developers have used "*android.permission.*" prefix. Such incorrect permissions cause security exceptions during runtime. Mistakes committed by developers can be mitigated through testing but testing Android apps is a challenging task [11, 12, 13, 14]. Though recent advancement in tools and techniques for testing Android apps [15], configuration-aware testing techniques are hardly available for Android apps.

Configuration errors, including errors in an Android manifest file, can be detected either indirectly through testing apps or directly through analyzing configuration files. Configuration parameters in the Android manifest file capture various aspects of an app such as security, user interface, execution etc. but most of the available testing

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.41

techniques are designed to test only one aspect of an app. Given the challenges in testing Android apps and the limitations in testing configuration errors, it is likely that some configuration errors will go unnoticed during testing. A more reliable option is to directly analyze configuration files for errors. One widely used analysis technique is a rule-based error detection. However, major challenges in rule-based error detection are to define and manage effective rules [3].

One key goal of this paper is to automatically detect developer mistakes in the manifest file. Towards this goal, *one of our key contribution in this paper is a rule-based static analysis tool named ManifestInspector*. The tool analyzes an Android manifest file and generates a warning message if any configuration parameters violate the specified rules. The source code of the tool is available in public domain [16]. *We used the tool to perform the empirical study and it generated total 59,547 configuration errors containing 478 unique errors from 11,110 apps*. While many of those errors are simple misconfigurations which a developer should avoid, we also found several errors which have serious consequences. Only 2,373 apps, among studied apps, do not have any configuration errors. The complete report is publically available for download [17].

In some extent, developer mistakes can be prevented if the developer has prior knowledge of the types of mistakes committed. Towards this goal, *another key contribution in this paper is to report and study mistakes committed by developers in writing Android manifest files*. Mistakes can be as serious as those which manifest as crash during an app execution and may lead to uninstallation of the app. On the other hand, mistakes can also be those which may go unnoticed to both users and developers. For example, an app may not be available for download in some devices because a mistake in a configuration parameter gives false impression that the device requires certain hardware to run the app. *We performed an empirical study on Android manifest files of 13,483 real-world Android apps*. To the best of our knowledge, we are the first to report developer mistakes in writing the manifest file based on a large scale empirical study.

The remainder of this paper is organized as follows. Section II presents an overview on the Android manifest file and discusses related works. Section III presents an overview of the ManifestInspector tool. Section IV presents key empirical results on developer mistakes in writing the manifest file. In Section V, we discuss key reasons behind developer mistakes. We also discuss threats to validity and limitations in Section V. Finally, Section VI concludes the paper.

II. BACKGROUND

A. Android Manifest Overview

Android manifest [18] is an essential configuration file in each Android app. Android system must have the file along with the app before it runs any app's code. The file is written manually by developers in XML format. The structure of the manifest file is shown in Figure 1. Currently, it has total 26 different elements with 175 different attributes or configuration parameters. These configuration parameters control various aspects of an app such as execution, performance, compatibility, security, etc. Some parameters come into play as early as apps get installed into Google Play store while many parameters play role only during apps execution.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
   <uses-permission />
   <uses-permission-sdk-23 />
   <permission />
   <permission-tree />
   <permission-group />
   <instrumentation />
   <uses-sdk />
   <uses-configuration />
   <uses-feature />
   <supports-screens />
   <compatible-screens />
   <supports-gl-texture />
   <application>
        …
   </application>
</manifest>
```

```
<application>
  <activity>
    <intent-filter>
      <action />
      <category />
      <data />
    </intent-filter>
    <meta-data />
  </activity>
  <activity-alias>
    <intent-filter> . . . </intent-filter>
    <meta-data />
  </activity-alias>
  <service>
    <intent-filter> . . . </intent-filter>
    <meta-data/>
  </service>
  <receiver>
    <intent-filter> . . . </intent-filter>
    <meta-data />
  </receiver>
  <provider>
    <grant-uri-permission />
    <meta-data />
    <path-permission />
  </provider>
  <uses-library />
</application>
```

Figure 1. Manifest file structure

One key function of the manifest file is to control execution behavior of an app's components. Android apps are composed of four kinds of components: activities, services, broadcast receivers, and content providers. These components perform distinct tasks. All the components of an app, which the system can launch, must be declared in the manifest file through <*activity*>, <*service*>, <*receiver*>, or <*provider*> elements. The component elements must be declared inside the <*application*> element. The components which are not declared inside the <*application*> element cannot be launched by the system. The attributes in the component elements and the application element define properties for the components and the app respectively. Some attributes can be defined for both app and components. In such cases, if properties defined by an attribute in application and component elements are different then the properties of the component overrides the properties of the application.

In Android apps, a component can be launched by other apps. Such components are called exported components and the capability must be declared by the components in the manifest file. A component can be explicitly exported by setting its *android:exported* attribute to *true*. If the attribute is not defined then the default behavior depends on <*intent-filter*> elements. Except content provider components, presence of one or more <*intent-filter*> elements inside a component element implicitly exports the component. The <*intent-filter*> elements advertise capability of a component to handle intents sent by other apps. The capability is

specified through <*action*>, <*category*>, and <*data*> sub-elements of an <*intent-filter*> element.

An Android manifest file plays vital role in implementing permission-based security at the application level. Permissions are defined, declared, and enforced through various elements and attributes of the manifest file. In Android apps, sensitive system resources are protected with system-defined permissions but sensitive app resources such as exported components should be protected with permissions defined by developers. Such custom permissions are defined through <*permission*> elements. The defined permissions can be enforced at both application level and component level through an *android:permission* attribute of an <*application*> element and component elements (<*activity*>, <*service*>, <*receiver*>, or <*provider*>) respectively. An app willing to access resources protected by permissions must declare the permissions through <*uses-permission*> or <*uses-permission-sdk-23*> elements.

In addition to the core app behavior, configuration parameters of the manifest file can affect an app's external entities such as Google Play store. Android apps are distributed through central channels usually called stores. Though existence of third-party stores, majority of Android apps are distributed through official Google Play store. Some configuration parameters of the manifest file play crucial role in availability of apps in the Play store. Google Play store decides whether an app is compatible with the devices based on some configuration parameters. Users can view and download only those apps from the Play store which are compatible with their devices.

An Android manifest is a single file but its configuration parameters control various internal as well as external entities of an app. Any mistakes in the manifest file can have huge impact on security, reliability, and availability of apps.

B. Related Works

Mistakes committed by developers in writing Android manifest files have never been studied before primarily. However, some studies have been performed, specifically targeting security aspect of Android apps. ComDroid [19] examines Android manifest files and generates a warning message if the exported components are not protected with permissions. Corpus of other studies [9, 20, 21, 22, 23] investigate the use of permissions against exported components. Barrera et al. [24] presented a methodology for the empirical study of permission-based security using a Self-Organizing Map. In the study, they mainly observed two kinds of mistakes. Developers used duplicate permissions and they requested permissions that do not exist. Separate investigations by Au et al. [25], Bartel et al. [26], and Felt et al. [27] discovered that developers are using more permissions (over-privileged) than required by apps. Through manual inspections, Watanabe et al. [28] observed that over-privileged apps are the result of developer mistakes in writing the manifest file. In comparison to these techniques, we specifically analyze Android manifest files and report all kinds of mistakes, including security-related mistakes. Reporting over-privileged apps require source code analysis which we currently do not perform.

The only tool available for detecting errors in Android manifest files is an Android lint [29]. It is a static analysis tool which performs analysis on entire source files of an app, including an Android manifest file. It checks for potential errors as well as various performance, security, usability, accessibility, and other issues for optimization. In comparison to ManifestInspector, which only analyzes the Android manifest file, lint performs analysis on entire source files which is a significant benefit and it results into another major advantage. Values of some configuration parameters in the manifest file depend on other source files. The correctness of those values can only be verified if a tool perform analysis on entire source files. Despite several significant advantages of using lint, ManifestInspector performs better when it comes to detecting errors in the Android manifest file. The sole reason is the number of effective rules defined by ManifestInspector. Currently, lint (in Android Studio 1.5) defines only 30 rules related to an Android manifest file whereas ManifestInspector defines 116 rules. ManifestInspector has 11 rules overlapped with the lint. We have excluded 8 rules specified by the lint which are source code dependent. We have also excluded remaining 11 rules defined in the lint which are less impactful.

In sharp contrast to Android apps, configuration errors have been well studied in system software. Yin et al. [30] performed characteristic studies of real-world configuration errors. Some of their findings about the cause of misconfigurations are consistent with our empirical results discussed in Section IV. Xu et al. [31] investigated the complexity of configuration due to large number of configuration parameters and their value space. They further studied the effectiveness of configuration simplification and configuration navigation approaches in reducing the complexity. Exhaustively testing large number of configuration parameters is expensive and may be infeasible in practice. The problem is addressed by various researchers [32, 33, 34, 35, 36] through combinatorial interaction testing, configuration prioritization, and symbolic evaluation techniques. Researchers have also proposed tools and techniques [37, 38, 39, 40, 41, 42, 43] to detect and fix configuration errors. In comparison to these configuration error detection techniques, our tool uses a rule-based error detection technique which requires domain specific customizations.

III. MANIFESTINSPECTOR – A RULE-BASED STATIC ANALYSIS TOOL

ManifestInspector is a rule-based static analysis tool for detecting errors in Android manifest files. The tool is written in Java programming language. It is available as an open source tool [16]. An overview of the ManifestInspector tool is shown in Figure 2. It takes an Android manifest file as input then it parses the manifest file using a DOM parser and

extracts structural information as well as values of attributes. The correctness of the extracted information is then verified against the stored valid information through predefined rules. For simplicity, we have used text files to store the valid information which can be easily edited. If the tool detects any violations of the rules then it reports the violations as warning messages.

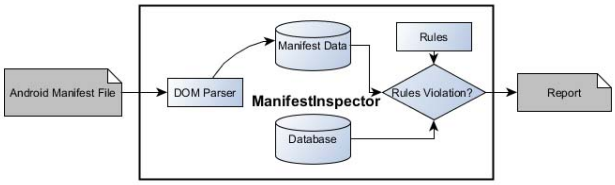

Figure 2. ManifestInspector overview

The ManifestInspector tool currently defines 116 rules which can be mainly classified into three categories. First category of rules verify organization of elements and attributes within the manifest file. This directly relates to the syntax of the manifest file. The rules specifically check two types of mistakes. Misplaced elements and attributes defined by the system and presence of elements and attributes which are not defined by the system. Second category of rules verify values of attributes. In addition to incorrect values, the rules also check empty values. Lastly, third category of rules verify correlation and dependency among attributes and elements. We extracted all the rules manually by going through the specification document of the Android manifest file [18].

One integral part of the tool is a database in the form of text files. One major advantage of using text files as a database is that even novice developers can easily edit the data. Developers may have to edit the data because the elements and attributes of the manifest file may be added or removed with the release of new versions of Android. The database stores all the valid elements and attributes of the manifest file. For each element, it stores valid attributes the element can contain and valid child elements. The database also stores valid attributes values.

The tool has been evaluated on Android manifest files of 13,483 real-world Android apps and the key evaluation results are discussed in Section IV. Out of 116 rules defined by the tool, the apps in the dataset violated 75 rules. The tool found total 59,547 possible developer mistakes with 478 unique mistakes in 11,110 apps. The tool did not find any mistakes in 2,373 apps.

IV. EMPIRICAL RESULTS

The empirical study was performed on 13,483 free Android apps downloaded from Google Play store during May and June of 2015. We downloaded top 500 free apps from each category displayed in the Play store. Some categories had listed less than 500 top free apps. The Play store was localized to United States using Tor browser [47]. After downloading all the apps which contained total 13,944 apps, we removed duplicate apps which were listed in more than one categories. We also removed duplicate apps with different versions. The downloaded apps were in APK (Android Application Package) format. We used Apktool [48] to extract the manifest file of each app. We then performed analysis on those extracted manifest files of 13,483 apps using our tool ManifestInspector. The tool found total 59,547 misconfigurations among 11,110 apps.

An overview of reported misconfigurations is shown in Figure 3. The reported misconfigurations are categorized into incorrect attribute values, misplaced attributes, incorrect attribute names, misplaced elements, incorrect element names, incorrect correlation and dependency among attributes and elements, and others. The others category include misconfigurations such as deprecated elements and duplicate elements. The impact of these misconfigurations on apps are shown in Figure 4. As shown in the figure, the level of impact has been classified into high, medium, and low. The high level misconfigurations directly affect functioning of apps. Some of the functionalities of apps will not perform correctly if the high level misconfigurations are not corrected. The medium level misconfigurations do not directly affect functioning of apps but these misconfigurations will affect overall quality of apps. For example, incorrectly implemented *<supports-screens>* elements may affect UI of apps when displayed on different screen size devices. The low level misconfigurations do not directly affect apps. However, developers need to aware of these misconfigurations. For example, we found several attribute names used by developers which are not part of the manifest file but they should be declared in other XML files. The level of impact for misplaced attributes category has not been shown in the figure because it requires analysis of elements in which the misplaced attributes should be declared. However, we manually checked some of the apps and found that these misconfigurations can severely affect apps. For example, we found 25 apps in which an *android:permission* attribute has been declared in *<intent-filter>* elements instead of *<receiver>* elements.

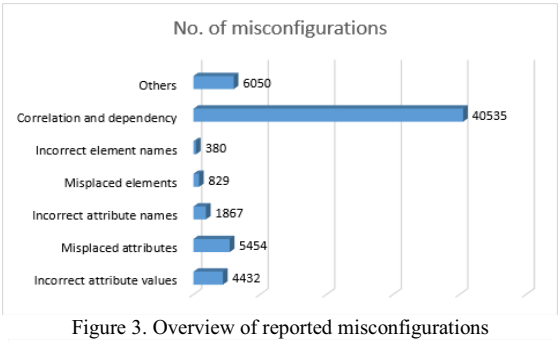

Figure 3. Overview of reported misconfigurations

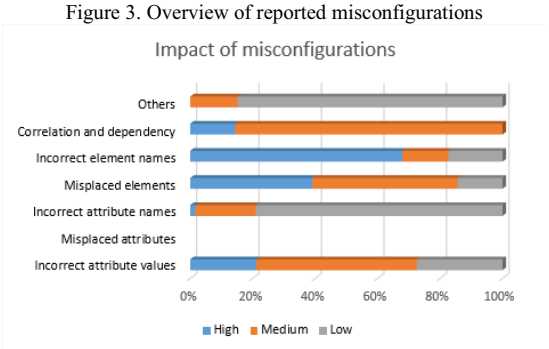

Figure 4. Impact of misconfigurations

In this paper, we discuss only those developer mistakes or misconfigurations which are either present in large quantity or have high significance. Interested readers can download the complete report [17]. The mistakes are classified into five major categories based on an app's areas which they affect: user interface, performance, execution, security, and compatibility.

A. App User Interface

Android TV home screen banner. A banner in an app can be implemented through an *android:banner* attribute. The attribute can be used in <*application*> and <*activity*> elements for default banner in all activities and a specific banner in an activity respectively. An activity represents a user screen in Android apps. The banner is used to represent an app in an Android TV home screen. An app intend to run on TV devices must declare a *CATEGORY_LEANBACK_LAUNCHER* intent filter in an activity. Thus, a dependency exists between the Android banner and the intent filter. Declaring an *android:banner* attribute is useless without declaring a *CATEGORY_LEANBACK_LAUNCHER* intent filter because the app will not be considered as a TV app. On the other hand, if the intent filter is used without the *android:banner* attribute then the TV home screen will not display the app. A developer must declare both the banner and the intent filter in a TV app. 170 apps such as *com.asg.hangerfree_1.94* and *com.appquiz.smart.games_2.20* in our dataset declare *android:banner* attributes without declaring the intent filter. On the other hand, 301 apps such as *com.babytv.LearningGames_1.42* and *com.autodesk.tinkerplay_1.0.1* declare *CATEGORY_LEANBACK_LAUNCHER* intent filters without declaring the *android:banner* attribute. These rules are also defined by the lint tool.

Up navigation. All screens, except home screen, of an app should offer users a way to navigate to the logical parent screen by pressing the Up button in the action bar. The feature is implemented by setting the *android:parentActivityName* attribute of <*activity*> elements. The value of *android:parentActivityName* represents a logical parent activity and it must be declared in the manifest file using an <*activity*> element. We found 94 activity components in 55 apps such as *com.bankrate.auto_1.06* and *com.booking_8.0.2* which declare *android:parentActivityName* attributes but the apps fail to declare the logical parent activity in the manifest file.

Task re-parenting. A task is a collection of activities with which users interact when performing a certain job. Tasks are managed by Android system. However, certain behavior of a task can be controlled by setting some attributes of <*activity*> elements. One such attribute is an *android:allowTaskReparenting*. It controls movement of an activity from the task that started it to the task it has an affinity for when the task is next brought to the front. Effectiveness of this attribute depends on the launch mode of the activity. An activity with a launch mode *singleTask* or *singleInstance* can only be at the root of a task so re-parenting is only allowed in *standard* and *singleTop* launch modes. The dataset has 228 activities in 203 apps such as *com.boxedup_1.1* and *com.cfinc.coletto_1.7.10* which use task re-parenting with a launch mode *singleTask* or *singleInstance*.

Theme. A default custom theme for all the activities and a custom theme for a specific activity can be implemented through *android:theme* attributes of <*application*> and <*activity*> elements respectively. The theme implemented through an <*activity*> element overrides the theme implemented through an <*application*> element. If the *android:theme* attribute is not set in <*activity*> or <*application*> elements then the activities use the default system theme. We found 1,081 apps such as *com.chili.monstertruck1_2.1* and *com.ciegames.RacingRivals_4.0* in which custom themes have been implemented through incorrect elements such as <*manifest*>, <*uses-library*>, <*meta-data*>, <*activity-alias*>, and <*service*>. These apps will use the default system theme instead of the defined custom themes.

Activity's main window versus soft keyboard. The state of a soft keyboard and the adjustment made to an activity's main window with respect to the soft keyword can be controlled using an *android:windowSoftInputMode* attribute of <*activity*> elements. Developers must use system-defined state or adjust values for the attribute. The value can be one state or adjust value or it can be a combination of one state value and one adjust value separated by a vertical bar (|). Setting multiple state or adjust values has undefined results. We found 58 activities in 10 apps such as *com.creditkarma.mobile_1.2.3* and *com.fingersoft.hillclimb_1.17.7* which have multiple state or adjust values. We also found 636 incorrect state or adjust values in 315 apps such as *com.frogmind.badland_1.7173* and *com.gismart.guitar_2.3.0*. The attribute *android:windowSoftInputMode* can be used only in <*activity*> elements but it has been used in several other elements such as <*manifest*>, <*application*>, <*activity-alias*>, and <*provider*> in 131 apps.

B. App Performance

Large heap. The attribute *android:largeHeap* of an <*application*> element specifies whether an app's process should be created with a large Dalvik heap. Setting the attribute value to *true* can only guarantee the increase in memory if the device has sufficient memory available. The attribute is effective only when declared in an <*application*> element. We found 59 apps such as *com.linhnv.apps.memecreator_1.3* and *com.lybrate.phoenix_2.1.6* which declare the attribute in incorrect elements such as <*manifest*>, <*supports-screens*>, <*activity*>, and <*service*>.

Hardware acceleration. A hardware-accelerated OpenGL renderer is available to apps starting from Android API level 11. It is enabled by default starting from API level 14 but it can be controlled by developers using an *android:hardwareAccelerated* attribute of <*application*> and <*activity*> elements. The attribute declared in other elements cannot have any effects on the default behavior. The dataset has 219 apps such as

com.magnetic.openmaps_4.39 and *com.mailboxapp_2.0.3* which declare the attribute in incorrect elements such as <*manifest*>, <*permission*>, <*supports-screens*>, <*service*>, and <*receiver*>.

C. App Execution

Components declaration. Each component of an app must be declared in the manifest file using <*activity*>, <*service*>, <*receiver*>, or <*provider*> elements. The immediate parent of these elements must be an <*application*> element. The components which are not declared or incorrectly declared in the manifest file cannot be launched by the system. Table 1 shows the components which are declared inside incorrect elements in several apps such as *com.miniclip.soccerstars_2.0.1* and *com.mobigrow.bankescape_1.0*.

Table 1. Components declared incorrectly

Component type	Incorrect parent elements	# apps	# components
<activity>	<manifest>	30	67
<service>	<manifest>	36	55
<receiver>	<manifest>	59	66
<provider>	<manifest>	1	1
<activity>	<activity>	2	2
<service>	<activity>	3	3
<receiver>	<activity>	2	2
<receiver>	<service>	1	1
<receiver>	<receiver>	1	2
<service>	<receiver>	1	1

Duplicate components. Single instance of all the components of an app must be declared in the manifest file. A component is uniquely identified by its class name which is also used as a value of an *android:name* attribute in <*activity*>, <*service*>, <*receiver*> and <*provider*> elements. Each component can have unique execution behavior which is defined through its various attributes. Duplicate components with different attributes can cause unexpected behavior during an app execution. We found 599 apps such as *com.noodlecake.anothercasesolved_1.3.2* and *com.nextmedia.gan_2.1.0* in the dataset which declare duplicate components.

Handling configuration changes at runtime. When a configuration change occurs at runtime, an activity gets shut down and restarted by default. The default behavior can be overridden by setting an *android:configChanges* attribute of <*activity*> elements. If a configuration listed in an *android:configChanges* attribute changes at runtime then the activity keeps running and calls *onConfigurationChanged()* method. Any mistakes in declaring an *android:configChanges* attribute brings the activity to its default behavior. We found 121 cases in 116 apps such as *com.ratrodstudio.skateparty2lite_1.12* and *com.onteca.CannonDefense_3.8* where the attribute is declared as *configChanges* instead of *android:configChanges*. We also found 50 cases in 15 apps such as *com.speedway.mobile_2.1* and *com.passenger.mytaxi_4.80.22* where the attribute values are empty. In addition to these mistakes, we found 363 cases where the attribute is declared in incorrect elements such as <*manifest*>, <*application*>, <*meta-data*>, <*activity-alias*>, <*service*>, <*receiver*>, and <*action*>.

Events handled through dynamic broadcast receivers. Broadcast receiver components handle system as well as apps generated broadcast events. Like other components, broadcast receiver components need to be registered in the manifest file but, unlike other components, it can also be declared dynamically. In fact, some system events can only be handled by dynamic broadcast receiver components. Broadcast receiver components declared in the manifest file cannot handle such events. Table 2 shows the system events which must be handled through dynamic broadcast receivers but developers have handled them incorrectly in several apps such as *com.trendmicro.tmmspersonal_6.0* and *com.iconnect.app.globalthemeshop_1.9*.

Table 2. System events incorrectly handled through static receiver instead of dynamic receiver components

System Events (actions)	# apps	# static receiver components
BATTERY_CHANGED	9	20
CONFIGURATION_CHANGED	3	8
SCREEN_ON	44	54
SCREEN_OFF	35	41
TIME_TICK	3	9

Intent filters without actions. An exported component advertises its capability through <*intent-filter*> elements in the manifest file. Another app can request the exported component to perform a task through an intent object. An intent describes an operation to be performed. A request to perform a task can only be processed by the component if the requested task matches with the advertised task. An intent filter advertises a task using <*action*>, <*data*>, and <*category*> elements. An <*action*> element specifies an action to be performed. An intent filter must contain at least one action otherwise intents with actions cannot go through the intent filter. It means the component cannot receive requests to perform a real task. The dataset has 1,270 intent filters in 351 apps such as *com.intuit.quickbooks_3.8* and *com.kiwi.enemylines_2.3.7* which do not contain an <*action*> element. In other 29 cases such as *com.lunagames.jurassicvr_1.1.0* and *com.miniclip.extremeskater_1.0.7*, the value of the *android:name* attribute of an <*action*> element is empty. It implicates the similar behavior as an intent filter without an action.

Wrong data format in intent filters. A <*data*> element of an intent filter specifies the data on which tasks performed by a component can operate on. It is specified through various attributes. The specification can be a data type (MIME type), a URI, or a combination of a data type and a URI. A URI is specified by separate attributes for each of its parts as *scheme://host:port[path|pathPrefix|pathPattern]*. The attributes of a URI are optional but mutually dependent. For example, if schema is not specified then all other URI attributes are ignored. Similarly, if host is not specified then port and all path attributes are ignored. According to the Android specification, a <*data*> element must contain a MIME type or a schema but our dataset has 30 <*data*> elements in 7 apps such as *com.touchsurgery_4.5.1* and *com.univision.android_1.0.13* which contain neither a MIME

type nor a schema. We also found 222 <*data*> elements in 187 apps which specify port or path attributes without specifying a host attribute.

Non-exported components with intent filters. A component can be used only by the declaring app or it can be used by other apps depending on whether the component is exported or not. The behavior is defined by using an *android:exported* attribute of the component. If the attribute value is *true* then the component is exported and it can be used by other than the declaring app. On the other hand, if the value is *false* then the component can be used only by the declaring app or the apps which have same user id as the declaring app. If the attribute is not declared explicitly then the default behavior depends on whether the component contains an intent filter. Presence of one or more intent filters implicate that the component is exported. We found 1,441 activity components in 360 apps, 656 service components in 417 apps, and 1,368 receiver components in 880 apps which explicitly set the *android:exported* attribute to *false* but contain intent filters.

In strict specification terms, declaring intent filters in non-exported components is incorrect but it has one key practical usability. Intent filters can receive implicit intents not only from other apps but also from the system but, unlike apps, system generated implicit intents can be received by non-exported components too. In fact, it is highly advised to make the components, which receive only system events, non-exported due to known security vulnerabilities [19]. Currently, ManifestInspector cannot specifically recognize system events so we have reported all non-exported components with intent filters as violations.

Exported components without intent filters. A component can be launched through intent objects explicitly by specifying its class name and implicitly by specifying properties of a task performed by the component. An exported component is not supposed to be launched explicitly by other apps because they don't have the component's class name. On the other hand, an implicit intent cannot be delivered to the components which do not declare intent filters. It means exported components without intent filters will behave almost similar to non-exported components with additional security risk. Adversaries can get the component's class name by reverse engineering the app and then they can send malicious intents explicitly. Thus, it is advised to make such components non-exported. We found 2,007 activity components in 1,164 apps, 426 service components in 314 apps, and 124 receiver components in 96 apps which are exported without intent filters.

D. App Security

Defining permissions. Permissions in Android apps can be categorized into system permissions and custom permissions. Sensitive system resources are protected with system permissions which are pre-defined by the system. Developers do not have to define them in the manifest file. On the other hand, sensitive app resources are protected with permissions defined by developers in the manifest file. Developer-defined permissions are called custom permissions. Permissions are defined using <*permission*> elements whose immediate parent element must be <*manifest*>. In the dataset, we found 63 system permissions defined in 46 apps such as *coo.videokikme.android_1.0.3* and *org.adw.launcher_1.3.3.9*. Defining system permissions does not have any serious implications. One mistake which does have serious implication is defining custom permissions in inappropriate places. We found 8 custom permissions defined in 8 apps whose immediate parent element is <*application*> instead of <*manifest*>. Resource protected with such incorrectly defined custom permissions are indeed unprotected.

Enforcing permissions – unprotected components. Custom permissions are used to protect apps' resources. One key apps' resource is a component. A component may perform sensitive tasks such as recording phone calls. Such a component, if exported, must be protected against unauthorized access. At the component level, permissions are enforced through an *android:permission* attribute of <*activity*>, <*service*>, <*receiver*>, and <*provider*> elements. At the application level, permissions are enforced through an *android:permission* attribute of the <*application*> element. We found 799 cases in 763 apps such as *com.adrenalinecrew.RSF2_1.4* and *com.akadilabs.airbuddy_2.5.1* where the *android:permission* attribute is used in incorrect elements such as <*action*>, <*intent-filter*>, <*manifest*>, and <*uses-sdk*> resulting into unprotected components or apps.

Non-exported components are well-protected because they are confined within an app execution space. On the other hand, exported components can be accessed by any apps including malicious apps. Thus, the exported components which perform sensitive tasks must be protected with permissions. In strict terms, the exported components which does not perform sensitive tasks should also be protected because those components may become accessories in accessing protected sensitive components. The vulnerability is widely known as privilege escalation attack [9, 21]. As shown in Table 3, large number of exported components are unprotected in the dataset. The dataset also has non-exported components protected with permissions as shown in Table 4 which indicates that either the developers don't know how to implement security or they are overcautious.

Table 3. Unprotected components

Exported component type	# apps	# unprotected exported components
Service	1105	1793
Receiver	6469	15825
Provider	1175	1421

Table 3 reports all unprotected exported components but, in reality, these exported components may not perform sensitive tasks or act as accessories in privilege escalation attack. Unnecessarily protecting exported components may jeopardize inter-app communication model [19] which is widely used in Android. This is one of the reasons why we have not reported unprotected activity components. Inter-app communication is mainly achieved through activity components. Another key reason to exclude activity components is the level of security risk. Activity components

represent UI screens with which users interact. An observant user can identify malicious behavior during interaction so activity components possess minimum security risk in comparison to other components which can perform tasks without user intervention.

Table 4. Non-exported components protected with permissions

Non-exported component type	# apps	# non-exported protected components
Service	256	421
Receiver	21	53
Provider	53	70

Declaring permissions. An app willing to access permission-protected resources must declare the permissions through an *android:name* attribute of <*uses-permission*> or <*uses-permission-sdk-23*> elements. The declared permissions are granted by users during install-time or run-time depending on the Android versions. Any mistakes in declaring permission names implicate that the app accesses resources without declaring the required permissions. If such an app tries to access the resource during execution then a security exception is thrown. We found 826 incorrect permission names in 603 apps, mostly caused by typos. Some incorrect permissions originated from developer's guess such as *ACCESS_LOCATION* instead of *ACCESS_COARSE_LOCATION* or *ACCESS_FINE_LOCATION* in *com.androidapplication.geeksquad.gsa.one_2.1* app. Some also originated from an incorrect prefix (package structure) used by system permission names. Most of the system permission names are defined with "*android.permission.*" prefix but there are other prefixes too. For example, we found *android.permission.INSTALL_SHORTCUT* permission name instead of *com.android.launcher.permission.INSTALL_SHORTCUT* declared by several apps such as *com.bestfreegames.goat_2.2.6* and *com.GavvaGames.ShareLand_1.050*.

An incorrect permission name wasn't the only problem we encountered when it comes to declaring permissions. We also found several mistakes in how and where the permissions are declared. Permissions are declared using <*uses-permission*> or <*uses-permission-sdk-23*> elements whose immediate parent element must be <*manifest*>. As shown in Table 5, incorrect elements as well as incorrect parent elements were used for declaring permissions in the dataset. These mistakes have same implication during app execution as declaring incorrect permission names. One mistake which does not have any severe implications is declaring duplicate permissions. We found 3,290 duplicate permissions declared in 1,116 apps.

Table 5. Permissions declared incorrectly

Incorrect elements	Incorrect parent elements	# apps	# elements
<android:uses-permission>	-	212	240
<user-permission>	-	5	6
<use-permission>	-	1	1
-	<application>	41	94
-	<activity>	1	2
-	<intent-filter>	2	2

Like sensitive resources, some sensitive actions are protected with system-defined permissions. An app cannot receive those protected actions without declaring the required permissions. It means the app will not be able to perform the intended task if the required permissions are not declared. Table 6 shows the protected actions used by the apps in the dataset without declaring the required permissions. For example, *com.gerth.Zoo_Scratch_1.95* app declares an *ACTION_BOOT_COMPLETED* action without declaring the *RECEIVE_BOOT_COMPLETED* permission. Similarly, *com.google.android.gm* app declares an *ACTION_DEVICE_ADMIN_ENABLED* action without declaring the *BIND_DEVICE_ADMIN* permission.

Table 6. Protected actions used without declaring the required permissions

Protected actions	Missing required permissions	#apps
ACTION_BOOT_COMPLETED	RECEIVE_BOOT_COMPLETED	560
ACTION_DEVICE_ADMIN_ENABLED	BIND_DEVICE_ADMIN	76
ACTION_SET_ALARM	SET_ALARM	5
ACTION_SET_TIMER	SET_ALARM	2
ACTION_NEW_OUTGOING_CALL	PROCESS_OUTGOING_CALLS	5
ACTION_PHONE_STATE_CHANGED	READ_PHONE_STATE	3
ACTION_STATE_CHANGED	BLUETOOTH	18
ACTION_DISCOVERY_FINISHED	BLUETOOTH	1
ACTION_DISCOVERY_STARTED	BLUETOOTH	1
ACTION_LOCAL_NAME_CHANGED	BLUETOOTH	1
ACTION_SCAN_MODE_CHANGED	BLUETOOTH	1
ACTION_ACL_CONNECTED	BLUETOOTH	1
ACTION_ACL_DISCONNECTED	BLUETOOTH	1
ACTION_ACL_DISCONNECT_REQUESTED	BLUETOOTH	1
ACTION_BOND_STATE_CHANGED	BLUETOOTH	1
ACTION_CLASS_CHANGED	BLUETOOTH	1
ACTION_FOUND	BLUETOOTH & ACCESS_COARSE_LOCATION	3
ACTION_NAME_CHANGED	BLUETOOTH	1

E. App Compatibility

Features used by apps. One of the important elements of the manifest file, which is largely undermined by developers, is a <*uses-feature*> element. This element has nothing to do with the Android system or an app's execution but it plays a key role in availability of apps on devices. The element has two key attributes: *android:name* and *android:required*. The *android:name* attribute specifies a hardware or software feature used by an app whereas the *android:required* indicates whether the specified feature is required by the app. The *android:required* attribute with *true* value indicates that the app cannot function if the specified feature is not present on devices. Google Play store filters such apps from users on devices which do not provide the required features. On the other hand, *false* value indicates that the app prefers to use the feature if present on the device but it can function without the specified feature. Google Play store does not filter such apps.

Developers may forget to declare features explicitly or they may use incorrect feature names in the *android:name* attribute. In such cases, Google Play store implicitly finds features required by apps. The Play store uses <*uses-permission*> elements as a main source for determining implicit features. For example, if an app uses an

ACCESS_WIFI_STATE permission and it does not explicitly declare an *android.hardware.wifi* feature then the Play store implicitly assumes that the hardware is required to run the app even the app can run without the hardware. Android documentation states that the absence of explicit declaration of a feature should be considered as an error. We found 12,287 instances of features which have not been declared explicitly in 7,615 apps such as *com.google.android.stardroid_1.6.4* and *com.gotv.crackle.handset_4.4.4.5*. We also found 283 cases in 209 apps such as *com.groupme.android_5.3.3* and *com.hybridforge.oppl_1.0.4* where incorrect feature names have been used. In addition to these mistakes, developers have used incorrect elements such as <*uses-featurea*>, <*usesfeature*>, <*use-feature*>, and <*used-feature*>. They have also used correct elements at incorrect places. The <*uses-feature*> element is declared inside <*manifest*> but it has been declared inside <*application*> in 36 cases among 19 apps. They have also used incorrect attributes such as *name* instead of *android:name* and *required* instead of *android:required* in 34 cases.

Screen orientation. An activity represents a user screen. The orientation of an activity during execution can be controlled by an *android:screenOrientation* attribute of <*activity*> elements. Values of the attribute are pre-defined by the system. Incorrect use of the attribute can cause unexpected execution behavior as well as compatibility issues. For example, if a developer declares one of the landscape or portrait values then it is considered as a hard requirement for the orientation. Consequently, the Play store filters the app on devices which do not support the orientation. We found 55 instances of incorrect attribute name *screenOrientation* in 53 apps and 79 incorrect values in 14 apps. We also found 388 cases in 340 apps where the *android:screenOrientation* attribute has been used in incorrect elements such as <*manifest*>, <*supports-screens*>, <*application*>, <*service*>, <*receiver*>, etc.

Screen configuration. Developers can specify screen configurations with which an app is compatible using <*screen*> elements inside a <*compatible-screens*> element. Google Play store uses this element to filter apps on devices which do not support the listed screen configurations. At least one instance of the <*screen*> element must be placed inside the <*compatible-screens*> element. Also, the <*screen*> element must include both *android:screenSize* and *android:screenDensity* attributes otherwise the element is completely ignored. We found 7 apps such as *com.backgammonlivefree_3.5.4* and *com.experian_1.2.1* which do not include even a single <*screen*> element inside a <*compatible-screens*> element. We also found 2 apps *com.myprograms.glasgow_3.0* and *com.exp.Doctor_at_home_3.0* where only one attribute is used in a <*screen*> element. The value of *android:screenDensity* is pre-defined by the system. In 447 cases, incorrect screen density values have been used.

V. DISCUSSION

A. Key Reasons behind Developer Mistakes

An Android manifest is a single file but the empirical results in Section IV clearly indicate that the mistakes in the manifest file can have huge impact on the security, reliability, and availability of apps. Developers cannot afford to ignore such mistakes. In Section IV, we studied the common mistakes committed by developers in writing the manifest file. The knowledge of common mistakes alone cannot prevent developers from committing the same mistakes. Developers must take measures against the root causes of those mistakes.

Misplacement of attributes and elements is clearly one of the leading problems in writing the manifest file. In most of the cases, frequency of a specific mistake is very high. For example, an *android:theme* attribute has been incorrectly used in the <*manifest*> element of 1009 apps. It is unlikely that the developers of all those apps committed human errors. One explanation is that the mistake may have originated from the human error but it must have propagated in large number of apps. Most of the Android developers learn from the official documentation and source code repositories. Clearly, the mistakes are propagating from the repositories with wrong source code. For example, a source code example for IBM Push Notification implementation has a mistake of using an *android:permission* attribute in <*intent-filter*> elements [44]. The mistake propagated into 25 apps such as *airborne.nbawp_3.1* and *cellfish.capamerica2_1.2*. Though human errors are difficult to prevent, precautions can be taken against error propagation. Rather than blindly following the wrong source code, developers should refer official documentation which is clear and concise about the attributes usages. One solution which can completely prevent the mistakes is automation of the manifest file. Though complete automation seems quite difficult, positions of attributes and elements within the manifest file can be automated.

During analysis, we observed same mistakes in several apps developed by the same software development house which can also be observed by interested readers in our complete report. It means developers are reusing the manifest file. Code reuse is prevalent in Android apps [45, 46]. Reuse of code is generally considered as a good practice under various circumstances. However, we would advise against the reuse of the manifest file because the file is highly customized according to an app's structure and requirements.

In addition to misplacement of attributes and elements, incorrect attribute and element names have been used in many apps. The attribute names in a manifest file are generally defined with "*android:*" prefix. In most of the cases, attribute names are missing the prefix. For example, developers have used *name* and *configChanges* instead of *android:name* and *android:configChanges* respectively. Incorrect element names have different mistake patterns than incorrect attribute names. For example, developers have used <*metadata*>, <*Application*>, <*support-screens*>, and <*android:uses-permission*> element names instead of

<*meta-data*>, <*application*>, <*supports-screens*>, and <*uses-permission*> respectively. Either the mistakes are caused by typos or developers have guessed the names which are incorrect. Again, the solution is to automate the manifest file so that developers do not have to type attribute and element names manually.

One leading cause of configuration errors in system software is correlation and dependency among configuration parameters. In this aspect, the Android manifest is not far from the system software. The empirical results in Section IV clearly indicate that the correlation and dependency among attributes and elements is a major cause of mistakes in writing the manifest file. Certainly, these properties increase the complexity which can be best handled with a tool support but developers can prevent these mistakes by carefully reading the documentation. Again, for some reasons, developers seem to be ignoring the documentation which clearly defines correlation and dependency for most of the attributes and elements.

Official documentation is a key source of information for Android developers. Despite the presence of well-organized information about the manifest file, the documentation is missing one key information. For most of the elements and attributes, the documentation provides information on their functionalities, correlation, and dependency. However, it fails to mention implications of their incorrect usages in several cases. Developers are likely to ignore mistakes in the manifest file unless they are aware of serious implications of those mistakes.

B. Threats to Validity and Limitations

The empirical study was performed on top 13,483 free Android apps downloaded from Google Play store. In comparison to more than 2.6 million apps currently available in the Play store, our dataset is very small. Though empirical results in this paper is valid for studied apps, it cannot be generalized for other Android apps. Further, the empirical study was performed only on top free apps. Lower ranking free apps as well as paid apps may have different results. However, correlation between quality of apps and their ranking in the Play store is yet to be established.

Android documentation does not provide a comprehensive list of system permissions. We obtained a list of system permissions by querying package manager on Android 6.0 but system permissions may be added or removed with the release of new versions of Android. Since many apps in our dataset target older versions of Android, we included the removed system permissions in the list. Still, the list may not be comprehensive. It can affect specifically two empirical results discussed in Section IV. First, number of system permissions defined by apps may increase and second, number of incorrect system permissions used by apps may increase.

While studying the use of incorrect permissions in Section IV, we only considered system permissions. Finding incorrect permissions requires knowledge of correct permissions. Getting comprehensive list of custom permissions is far more difficult than getting comprehensive list of system permissions. Each Android app may define custom permissions which means all the available apps must be analyzed to get the complete list of custom permissions. On top of this, each day hundreds of new Android apps are added in the Play store which makes the task virtually impossible.

As with other rule-based static analysis tools, ManifestInspector has some generic limitations. The rules defined by ManifestInspector is not comprehensive. We have defined the rules by reading official Android documentation. We may have missed some specifications or the documentation may be incomplete. Also, the tool does not analyze source code which means rules specifying configuration parameters dependency on source code cannot be implemented. ManifestInspector currently does not define any rules which are source code dependent. We consider this as one of our future task. Another major limitation is that the defined rules may become obsolete. Android specifications related to the manifest file may change with the release of new Android versions resulting into obsolete rules.

To avail some services provided by private service providers, developers need to customize the manifest file which is not in line with the official documentation. For example, integrating an app with Amazon Device Messaging requires an <*amazon:enable-feature*> element to be declared inside the <*application*> element. The ManifestInspector treats all the elements which are not specified by Android as developer mistakes. We found 101 apps in the dataset containing <*amazon:enable-feature*> elements.

VI. CONCLUSION

In this paper, we studied mistakes committed by developers in writing the Android manifest file. The empirical results clearly indicate that developers are indeed making mistakes in writing the manifest file. With some impact-less mistakes, developers are also making those mistakes which can have huge impact on security, reliability, and availability of the apps. Most of the developer mistakes can be classified into three main categories: misplaced elements and attributes, incorrect attributes values, and incorrect dependency and correlation among attributes and elements. Though, mistakes may have mainly originated from human errors, propagation of mistakes seems to be the major cause of their presence in the manifest file. Most of the mistakes can be prevented if the structure (place of elements and attributes) of the manifest file is automated. Developers can also mitigate some of the mistakes by using tools such as Android lint. We have also presented a tool called ManifestInspector which can help developers in identifying and mitigating these mistakes.

ACKNOWLEDGMENT

This research was supported by the BK21 Plus project (SW Human Resource Development Program for Supporting Smart Life – 21A20131600005) and Basic Science Research Program through the National Research Foundation of Korea (No. NRF-2014R1A1A2058733) funded by the Ministry of Education, School of Computer Science and Engineering, Kyungpook National University, Korea.

REFERENCES

[1] AppBrain - Number of available Android apps in the Play Store. http://www.appbrain.com/stats/number-of-android-apps.

[2] S.L. Lim, P.J. Bentley, N. Kanakam, F. Ishikawa, and S. Honiden, "Investigating country differences in mobile app user behavior and challenges for software engineering," IEEE Transactions on Software Engineering, 41(1), 2015, pp.40-64.

[3] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," ACM Computing Surveys (CSUR), 47(4), 2015, p.70.

[4] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," Proc. of the 6th International Workshop on Automation of Software Test, 2011, pp. 77-83. ACM.

[5] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," IEEE security & privacy, 7(1), 2009, pp.50-57.

[6] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," In USENIX security symposium, Vol. 2, 2011, p. 2.

[7] A.K. Jha and W.J. Lee, "Analysis of Permission-based Security in Android through Policy Expert, Developer, and End User Perspectives," Journal of Universal Computer Science, 22(4), 2016, pp.459-474.

[8] L. Davi, A. Dmitrienko, A.R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," International Conference on Information Security, 2010, pp. 346-360. Springer Berlin Heidelberg.

[9] A.P. Felt, H.J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-Delegation: Attacks and Defenses," USENIX Security Symposium, Vol. 30, 2011.

[10] StackOverflow - Android manifest error. http://stackoverflow.com/search?q=android+manifest+error.

[11] D. Amalfitano, A.R. Fasolino, P. Tramontana, and B. Robbins, "Testing Android Mobile Applications: Challenges, Strategies, and Approaches," Advances in Computers, 89(6), 2013, pp.1-52.

[12] A.I. Wasserman, "Software engineering issues for mobile application development," Proc. of the FSE/SDP workshop on Future of software engineering research, 2010, pp. 397-400. ACM.

[13] H. Muccini, A.D. Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," Proc. of the 7th International Workshop on Automation of Software Test, 2012, pp. 29-35. IEEE Press.

[14] M.E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 15-24. IEEE.

[15] S. Zein, N. Salleh, and J. Grundy, "A systematic mapping study of mobile application testing techniques," Journal of Systems and Software, 117, 2016, pp.334-356.

[16] ManifestInspector - A rule-based static analysis tool. https://github.com/HiFromAjay/ManifestInspector.

[17] ManifestInspector analysis report. https://github.com/HiFromAjay/ManifestAnalysisReport/blob/master/Android_Manifest_Analysis_Report.pdf.

[18] Android Manifest File. http://developer.android.com/guide/topics/manifest/manifest-intro.html.

[19] E. Chin, A.P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," Proc. of the 9th international conference on Mobile systems, applications, and services, 2011, pp. 239-252. ACM.

[20] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y.L. Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," Proc. of the 22nd USENIX security symposium, 2013, pp. 543-558.

[21] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," IEEE transactions on Software Engineering, 41(9), 2015, pp.866-886.

[22] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," Proc. of the 2012 ACM conference on Computer and communications security, 2012, pp. 229-240. ACM.

[23] L. Li, A. Bartel, J. Klein, and Y.L. Traon, "Automatically exploiting potential component leaks in android applications," 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2014, pp. 388-397. IEEE.

[24] D. Barrera, H.G. Kayacik, P.C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," Proc. of the 17th ACM conference on Computer and communications security, 2010, pp. 73-84. ACM.

[25] K.W.Y. Au, Y.F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," Proc. of the 2012 ACM conference on Computer and communications security, 2012, pp. 217-228. ACM.

[26] A. Bartel, J. Klein, M. Monperrus, and Y.L. Traon, "Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android," IEEE Transactions on Software Engineering, 40(6), 2014, pp.617-632.

[27] A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," Proc. of the 18th ACM conference on Computer and communications security, 2011, pp. 627-638. ACM.

[28] T. Watanabe, M. Akiyama, T. Sakai, H. Washizaki, and T. Mori, "Understanding the inconsistencies between text descriptions and the use of privacy-sensitive resources of mobile apps," Eleventh Symposium On Usable Privacy and Security (SOUPS), 2015, pp. 241-255. USENIX Association.

[29] Android Lint. http://developer.android.com/tools/help/lint.html.

[30] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L.N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, pp. 159-172. ACM.

[31] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software," Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 307-319. ACM.

[32] C. Yilmaz, M.B. Cohen, and A.A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," IEEE Transactions on Software Engineering, 32(1), 2006, pp.20-34.

[33] E. Dumlu, C. Yilmaz, M.B. Cohen, and A. Porter, "Feedback driven adaptive combinatorial testing," Proc. of the 2011 International Symposium on Software Testing and Analysis, 2011, pp. 243-253. ACM.

[34] X. Qu, M.B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," Proc. of the 2008 international symposium on Software testing and analysis, 2008, pp. 75-86. ACM.

[35] X. Qu, M. Acharya, and B. Robinson, "Impact analysis of configuration changes for test case selection," IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE), 2011, pp. 140-149. IEEE.

[36] E. Reisner, C. Song, K.K. Ma, J.S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," Proc. of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, 2010, pp. 445-454. ACM.

[37] M. Attariyan and J. Flinn, "Automating Configuration Troubleshooting with Dynamic Information Flow Analysis," OSDI, Vol. 10, 2010, pp. 1-14.

[38] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," Proc. of the 26th IEEE/ACM International Conference on

Automated Software Engineering, 2011, pp. 193-202. IEEE Computer Society.

[39] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, "Range fixes: Interactive error resolution for software configuration," IEEE Transactions on Software Engineering, 41(6), 2015, pp.603-619.

[40] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," Proc. of the 2011 USENIX conference on USENIX annual technical conference, 2011, pp. 28-28. USENIX Association.

[41] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection'" ACM SIGPLAN Notices, 49(4), 2014, pp.687-700.

[42] S. Zhang and M.D. Ernst, "Automated diagnosis of software configuration errors," Proc. of the 2013 International Conference on Software Engineering, 2013, pp. 312-321. IEEE Press.

[43] S. Zhang and M.D. Ernst, Which configuration option should I change?," Proc. of the 36th International Conference on Software Engineering, 2014, pp. 152-163. ACM.

[44] IBMPushNotification - a mistake in implementing permission. http://developer.xtify.com/display/sdk/Getting+Started+with+Google+Cloud+Messaging.

[45] I.J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A.E. Hassan, "A large-scale empirical study on software reuse in mobile apps," IEEE software, 31(2), 2014, pp.78-86.

[46] I.J.M. Ruiz, M. Nagappan, B. Adams, and A.E. Hassan, "Understanding reuse in the android market," IEEE 20th International Conference on Program Comprehension (ICPC), 2012, pp. 113-122. IEEE.

[47] Tor Browser - https://www.torproject.org/projects/torbrowser.html.en

[48] Apktool - https://ibotpeaches.github.io/Apktool/