Characterizing Android-specific crash bugs

Ajay Kumar Jha, Sunghee Lee, 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

***Abstract*—Android platform provides a unique framework for app development. Failure to comply with the framework may result in serious bugs. Android platform is also evolving rapidly and developers extensively use APIs provided by the framework, which may lead to serious compatibility bugs if developers do not update the released apps frequently. Furthermore, Android apps run on a wide range of memory-constrained devices, which may cause various device-specific and memory-related bugs. There are several other Android-specific issues that developers need to address during app development and maintenance. Failure to address the issues may result in serious bugs manifested as crashes. In this paper, we perform an empirical study to investigate and characterize various Android-specific crash bugs, their prevalence, root causes, and solutions by analyzing 1,862 confirmed crash reports of 418 open source Android apps. The investigation results can help app developers in understanding, preventing, and fixing the Android-specific crash bugs. Moreover, the results can help app developers and researchers in designing effective bug detection tools for Android apps.**

Keywords*—Android apps, crash bug analysis, mining crash bugs, characterizing crash bugs

I. 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 [1]. The platform is evolving rapidly and, at the same time, it has become pervasive. The diversity of devices such as TVs, infotainment systems, smartphones, and tablets are operating on Android, which runs apps with diverse application domains including critical domains such as finance and health. Therefore, the reliability and security of Android apps are the major concerns for app users [2].

Researchers have proposed several testing tools and techniques [3, 4] to improve the quality and reliability of Android apps. However, developers still prefer to test their apps manually due to various complications in using new tools and techniques [5, 6]. Manual testing of Android apps requires developers to have extensive knowledge of both the Java programming language and the Android framework. Furthermore, they need to have extensive knowledge of the nature of Java and Android-specific bugs, their root causes, and solutions. Although Android apps are mostly written in the Java programming language, developers have to extensively implement various callback methods or APIs (Application Programming Interface) provided by the Android framework. While the information on typical bugs related to the Java programming language and their solutions are readily available for stakeholders, there is a lack of comprehensive information on the Android-specific bugs and their solutions. One reason may be the rapid evolution of the Android platform such as the introduction of runtime permission, which results in new categories of bugs. While developers' discussion forums such as Stack Overflow are extremely useful in identifying and fixing the bugs, user-reported bugs may not have been discussed in the forums, which we found in several cases in this study. A comprehensive investigation and characterization of user-reported bugs of deployed apps and their solutions would help app developers in identifying, preventing, and fixing the real bugs. Furthermore, it would help researchers in identifying future research directions that could lead to more effective tools and techniques for testing Android apps.

To address this need, we perform an empirical study by mining bug reports of free and open source Android apps. Particularly, we perform the study on only those bugs that manifest as crashes. Our first goal of this empirical study is to understand the prevalence of Android-specific crash bugs in Android apps. Therefore, we manually identify the Android-specific crash bugs by analyzing 1,862 confirmed user-reported crash bugs from 418 open source Android apps. Our next goal is to characterize the identified Android-specific crash bugs. Therefore, we analyze the identified Android-specific crash bugs manually and place them in different categories based on their distinctive properties. Our final goal is to present the root causes of the Android-specific crash bugs and their solutions. To achieve the final goal, we analyze the identified Android-specific crash bug reports, developers' discussion about the crash bugs in the issue tracker, and the fixes implemented by developers. In this paper, we present the results of the investigations.

The major contributions of this paper can be summarized as follows:

- Performs a large-scale empirical study on user-reported crash bugs by mining 1,862 confirmed crash bug reports of 418 open source Android apps.
- Distinguishes and characterizes Android-specific crash bugs by analyzing the 1,862 user-reported crash bugs.
- Presents an extensive analysis of the root causes of Android-specific crash bugs and their solutions implemented by developers in fixing the bugs.

The rest of the paper is organized as follows. Section II presents related works. Section III describes our methodology of obtaining dataset and analyzing crash bugs. Section IV presents a characterization of Android-specific crash bugs and their root causes. Furthermore, in this section, we discuss different solutions implemented by developers in fixing the bugs. We discuss the empirical results in Section V. We discuss threats to validity in Section VI. Finally, the paper concludes in Section VII.

II. RELATED WORK

Researchers have performed several characterization studies on bugs in the Android platform and apps. Linares-Vásquez et al. [7] performed a taxonomy of Android bugs by analyzing 1,230 confirmed bugs including 430 Android-specific bugs. They also proposed a mutation testing framework for Android apps based on the taxonomy. Hu et al. [8] performed a bug mining study on 10 open source Android apps to understand the nature and frequency of bugs in Android apps. They also proposed a UI testing technique. In comparison to their studies [7, 8], we characterize only Android-specific crash bugs on a significantly large dataset. Furthermore, unlike their studies, we discuss the solutions implemented by developers in fixing the crash bugs. Fan et al. [9] performed a large-scale analysis of framework-specific exceptions in Android apps. Similar to our study, they characterize the framework exceptions and their fixes. However, the characterization in this study is substantially different from their study. We believe both works can be complementary to each other in thoroughly understanding the Android-specific crash bugs. Bhattacharya et al. [10] performed an empirical study on the bugs in the Android platform and 24 open source Android apps to understand the quality of bug reports and bug-fixing process. They also characterized security bugs. However, they do not discuss other types of Android-specific bugs and their solutions. Maji et al. [11] performed a characterization study on failures in Android and Symbian platforms. They also discussed the solutions implemented by developers in fixing the bugs. Unlike their study of platform-specific failures, we target Android-specific crash bugs in apps. Joorabchi et al. [12] performed an empirical study to characterize non-reproducible bugs in six different projects of desktop, web, and mobile domains. Unlike their study, we target only fixed Android-specific crash bugs. Pathak et al. [13], Liu et al. [14, 15], and Shahriar et al. [16] performed characterization studies on performance bugs in Android apps. In comparison to these studies [13, 14, 15, 16], our study targets all types of Android-specific crash bugs in deployed apps. Since the Android platform is evolving rapidly, the bugs originating from the features introduced in the later versions of the platform such as runtime permission and fragment have not been discussed by the most of the existing works. Furthermore, the existing works do not discuss the solutions implemented by developers in fixing various types of Android-specific crash bugs.

Researchers have also proposed several tools and techniques to detect [7, 8, 17, 18, 19, 20], reproduce [21, 22, 23, 24, 25, 26, 27], and repair [28] bugs in Android apps, which is not our goal in this paper. However, results of this study can help developers and researchers in designing effective bug detection tools for Android apps.

III. METHODOLOGY

In this study, our goal is to investigate and characterize only Android-specific crash bugs. Therefore, in this section, we first present how we collected the data for our study, then we separate Android-specific crash bugs from Java-related crash bugs. We considered a bug as an Android-specific crash bug if the bug stems from the misuse of Android framework or APIs.

A. Data Collection

Our study requires analysis of source code and bug reports. Therefore, we used F-Droid [29], 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 1,560 apps. Then, we performed a filtering process on the selected apps. First, we removed 171 apps that had no reported issues or bugs. Our study requires the analysis of solutions implemented by developers in fixing the bugs. Therefore, we further removed 179 apps that had no closed issues, resulting in a dataset of 1,210 apps.

In this study, we target only those bugs that manifest as crashes. Therefore, we searched the closed issues of each app with three different keywords: crash, exception, and force close. Among 1,210 apps, the search results did not produce any bug reports in 648 apps. Next, we manually analyzed the crash reports of the remaining 562 apps. However, we could not determine the root causes of the crash bugs in 144 apps due to one of the following reasons: the issue was closed without discussing the fix or providing the fix patch, the bug was already fixed in version X or will be addressed in the next release, the bug could not be reproduced, the bug was caused by compile related issues, the bug was in the development branch or debug version, the bug was not related to the app, the bug was resolved by re-installing the app or clearing the app data, the bug was not valid anymore, and the bug will not be fixed. After excluding the 144 apps with inconclusive crash reports, we had the final dataset of 418 apps that also had several inconclusive crash reports. However, each app in the final dataset had at least one conclusive crash report. Therefore, we excluded the inconclusive crash reports from the apps in the final dataset, resulting in total 1,862 confirmed crash reports in 418 apps.

Among the final dataset of 418 apps, 289 apps are available in the Google Play store, which have various download ranges as shown in Fig. 1. Overall, our dataset has diverse categories of apps including apps from the Google Play store and third-party app stores.

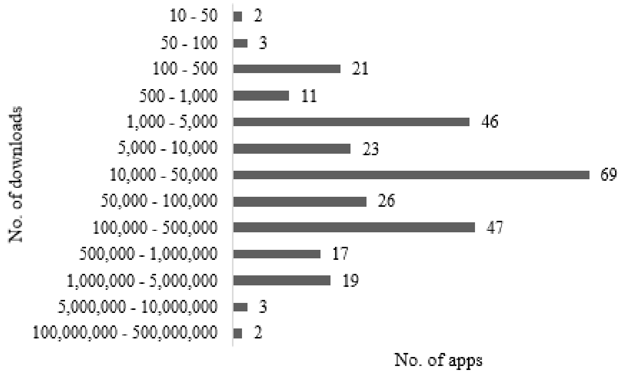

Fig. 1. Apps download ranges in the Google Play store.

B. Android-specific Crash Bugs

In this paper, our goal is to investigate and characterize only Android-specific crash bugs. Therefore, we manually analyzed the 1,862 confirmed crash reports using an open coding approach [31, 32] and differentiated between Java-related crash bugs and Android-specific crash bugs. The result provides information on distribution or prevalence of Android-specific crash bugs in Android apps. We identified 672 crash bugs as Android-specific crash bugs, which is 36% of the total crash bugs in our dataset. The result is similar to the result obtained by Mario Linares-Vásquez et al. in their study [7]. However, their dataset is smaller with 430 Android-specific bugs among 1,230 total confirmed bugs. In our dataset, the Android-specific crash bugs are distributed in 265 apps among 418 total apps. We have made the dataset available to enable other researchers to access and reproduce our study [30].

36% of the crash bugs are Android-specific crash bugs.

IV. CHARACTERIZATION OF ANDROID-SPECIFIC CRASH BUGS

To characterize the Android-specific crash bugs, the first and the second author independently analyzed the bug reports including bug description, developers' discussion about the bug, and solutions provided by the developers. The authors then independently categorized the bugs following an open coding approach [31, 32]. The disagreement of placing a bug in a category (6% of the bugs) was resolved through discussion between the first and the second author. The identified Android-specific crash bugs were placed in ten different categories as shown in Fig. 2. The figure shows the number of apps with the number of Android-specific crash bugs in each category. For example, the dataset has 59 permission-based security crash bugs in 46 apps. We have provided the complete data of Android-specific crash bugs in each category for interested readers [30].

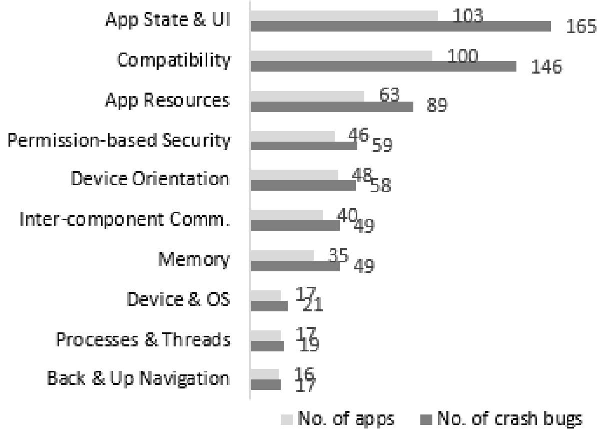

Fig. 2. Characterization of Android-specific crash bugs.

A. App State and UI Bugs

Android apps are composed of four types of components: activities, services, broadcast receivers, and content providers. An Android app state is basically representative of its components state, which is implemented through various callback methods provided by the platform. Moreover, Android apps are developed mainly with the UI-centric approach in which everything is built around the UI. Our dataset has 165 crash bugs in 103 apps caused by the incorrect implementation of app state and UI. Furthermore, our dataset has several other crash bugs related to app state and UI. However, we have placed them in separate categories considering how the crash bugs manifested during the execution of the apps. The distribution of the app state and UI crash bugs according to their origin in different parts of apps is shown in Fig. 3. App components such as activity, service, receiver, and provider have the largest number of crash bugs. However, in comparison to fragments that are part of activity components, the number of crash bugs in all the app components are not significantly high. Clearly, developers need to carefully implement fragments. A Context [33] that represents the global app state and a View that represents a user interface combined constitute 41% of the app state and UI crash bugs. However, considering their frequent usage during app development, the result is not surprising.

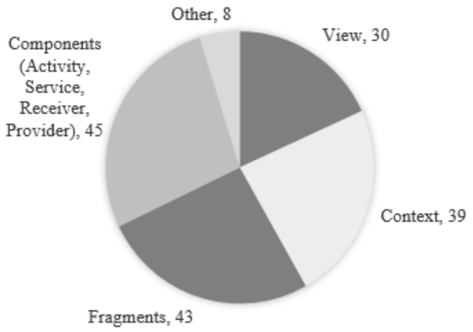

Fig. 3. Distribution of app state and UI bugs according to their origin.

Activity: An activity component that represents a single user screen is the most frequently used component in Android apps [34]. It has mainly four states: running, paused, stopped, and restored. The state transitions are handled through various callback methods. Our dataset has 24 crash bugs in 21 apps caused by incorrect handling of the state transitions. For example, accessing app or system resources in incorrect callback methods, not implementing *onNewIntent()* callback method when relaunching an activity while at the top of the activity stack, and not checking *isFinishing()* when dismissing dialogs after the activities that opened the dialogs have finished. Developers have fixed the crash bugs by accessing resources in appropriate callback methods, implementing new callback methods, and catching exceptions.

Service: Service components are generally used to perform long-running background tasks such as playing music. A component of an app can start a service component or bind to a service component. When a service component is started by calling *startService()*, the started service keeps running until it stops itself by calling *stopSelf()* or another component stops it by calling *stopService()*. However, if a component binds to a service by calling *bindService()*, the service runs as long as the component is bound to it. The bound service can interact with the component by offering a client-server interface until the component unbinds it by calling *unbindService()*. Our dataset has 13 crash bugs in 12 apps caused mostly by incorrect implementations of different callback methods. For example, calling *unbindService()* on the services that are no longer bound, interacting with the services that are no longer bound, and cleaning up resources that no longer exist before destroying or stopping the services.

Broadcast Receiver and Content Provider: Broadcast receiver components handle broadcast events generated by the system and apps. A broadcast receiver component can be created statically by declaring it in the Android manifest file and dynamically by registering it in the source code through the *registerReceiver()* method. The dynamically created receivers must be unregistered by calling the *unregisterReceiver()* method. However, developers should be careful when and where to call the *unregisterReceiver()* method. Our dataset has 5 apps that call the *unregisterReceiver()* method on the receivers that have already been unregistered, resulting in 5 crash bugs. Our dataset has also 1 crash bug in a receiver and 2 crash bugs in content providers caused by incorrect configurations in the Android manifest file.

Fragment: Starting from API level 11, fragments can be used to separate distinct elements of an activity, which define their own UI and lifecycle. Similar to activity components, state transitions in fragments are handled through various callback methods. Our dataset has 12 crash bugs in 12 apps caused by incorrect handling of the state transitions. Developers have fixed the bugs by using callback methods appropriately. Our dataset has also 2 crash bugs caused by fragments interacting with UI in incorrect states, 1 crash bug caused by instantiating an anonymous class fragment, and 1 crash bug caused by inflating a fragment within a fragment, which is only allowed programmatically. Fragments can be added, removed, or replaced to/from a running activity in response to user interactions. Each set of changes that are committed to the activity is called a transaction. To prevent state loss, Android does not allow to commit the fragment transactions after the activity has saved its state. Our dataset has 11 apps that commit fragment transactions after the activities have saved their states, resulting in 12 crash bugs. Developers have fixed the crash bugs by prohibiting the commits after the activities have saved their states. Our dataset has also 15 crash bugs in 12 apps caused by fragments not attached to their host activities. Developers have fixed most of the crash bugs by checking *isAdded()* on the fragments, checking null or *isFinishing()* on the activities, and catching the exceptions.

Context: In Android, a Context represents the context of the current state of an app or an object, which is used to access global information of the app environment. Furthermore, it is used for performing app-level operations such as launching activities, receiving intents, etc. A Context can represent an app context, an activity context, or a service context, which have different scopes in an app. Our dataset has 39 crash bugs in 30 apps caused by incorrect use of the context objects. Developers have used various techniques to fix the crash bugs. They have fixed 17 crash bugs by using contexts with appropriate scope such as using an app context instead of an activity context, 13 crash bugs by checking null on context objects, 5 crash bugs by retrieving the context correctly, 2 crash bugs by declaring the context with the final keyword, 1 crash bug by handling a null context, and 1 crash bug by using a singleton.

View and Action Bar: The user interface for each component of an app is defined using a hierarchy of View and ViewGroup objects. Our dataset has 3 crash bugs caused by defining multiple parent views for a child view. The View can be of different types. A ListView displays items in the list. The data in the list are populated through an Adapter by calling *setAdapter()*. Our dataset has 4 crash bugs caused by not notifying ListView about the data change in the Adapter. Furthermore, a header can be added in the ListView by calling *addHeaderView()*. However, the header view must be added before setting the adapter to the ListView. Our dataset has 2 crash bugs caused by setting adapter before adding a header view. An explicit view can be set in an activity using *setContentView()*. Furthermore, each activity can access features of its associated window through *requestWindowFeature()*. However, window features must be accessed before setting the content view. Our dataset has 3 crash bugs caused by accessing window features after setting the content views. A view of an activity is detached from the window manager when the activity is destroyed. However, a dialog started by the activity may still be running. Our dataset has 3 crash bugs caused by views not attached to the window manager. Our dataset has 7 other crash bugs in 6 apps caused by incorrect handling of views such as implementing ActionBarContextView incorrectly, performing a user action before the view is retrieved, using a custom view incorrectly, too many nested views causing StackOverflow, and getting views incorrectly. In addition to the View, Android provides various standard UI components such as action bars. All activities in an app that use the default system theme have an action bar. Developers can also implement an action bar using the support library's Toolbar class. However, an activity must have only one action bar. Our dataset has 4 crash bugs caused by using multiple action bars in an activity.

Other: Our dataset has 8 other crash bugs in 8 apps caused by various app state and UI issues such as incorrect use of wake lock and WIFI lock, exceeding maximum cursor limit, and incorrect implementation of menus.

App developers should 1) use appropriate callback methods in the appropriate order, 2) do not commit fragment transactions after the activity has saved its state, 3) check the activity has not been destroyed before performing operations on its fragments and dialogs, and 4) use contexts with the appropriate scope.

B. *Compatibility Bugs*

The Android platform provides several APIs and support libraries for app development, which makes it easier and faster to develop apps on the Android platform. However, due to the rapid evolution of the Android platform, developers need to update their apps frequently. Failure to update the APIs and support libraries in apps may result in serious compatibility bugs.

Since the launch of the Android platform, it has evolved to API level 28 in very short duration. Providing support for all the API levels or even for the target API level and below is a non-trivial task for app developers. Our dataset has 36 apps that use incompatible APIs, resulting in 47 crash bugs with

NoSuchMethodError exceptions. Developers have fixed 46 crash bugs by changing the implementation such as performing build version check in the source code before using the APIs, removing the APIs, or replacing the APIs. The remaining one crash bug was fixed by catching the exception. In addition to the crash bugs with the *NoSuchMethodError* exception, our dataset has one crash bug with *NoSuchFieldError* exception that was fixed by catching the exception, 3 crash bugs with *NoClassDefFoundError* exception that were fixed by using other classes or checking the build version, and 4 crash bugs with *UnsupportedOperationException* that were fixed by changing implementation or catching the exception.

In addition to the support libraries provided by the platform, developers can use various external libraries or dependencies in apps. Our dataset has 46 crash bugs in 42 apps caused by libraries incompatibilities. Developers have used various techniques to fix the crash bugs. They have fixed 27 crash bugs by upgrading the libraries, 5 crash bugs by downgrading the libraries, 3 crash bugs by removing the libraries, 1 crash bug by restoring a library, 1 crash bug by using a different class of the library, and 1 crash bug by upgrading a build tool version in the build.gradle file. In addition to upgrading, downgrading, removing, and reusing the libraries, developers have also fixed 8 crash bugs by providing workaround for the bugs in the libraries. For example, they have fixed 3 crash bugs by checking the build version, 1 crash bug by checking a null condition, 1 crash bug by implementing the missing feature, 1 crash bug by removing a method call, and 2 crash bugs by adding the required FileProvider class in the Android manifest file.

Our dataset has several other crash bugs caused by APIs incompatibilities. Some of the notable crash bugs and their fixes are shown in Table 1. Our dataset has a single instance of each crash bug shown in the table.

App developers should 1) check APIs support level and perform build version check in the source code before using the APIs, 2) avoid using unstable APIs and libraries, and 3) update APIs and libraries frequently.

TABLE 1. COMPATIBILITY CRASH BUGS

Crash Bugs	Fixes
bindService() does not support implicit intents starting with API 21.	Use an explicit intent.
Error inflating class fragment. Nested fragments were introduced in API 17.	Remove nesting.
the fragment.getlayoutinflater method name is a duplicate of a final method in class Landroid/app/Fragment.	Rename the method for API 26.
android:src for image view on vector is not supported below API 21.	Use app:srcCompat.
Fragment.onAttach(Context) is not called by android below API 23.	Override deprecated Fragment.onAttach(Activity).
Honeycomb version 11 does not show title bar (has custom title bar, so getActionBar() returns null).	Null check.
Providers are by default exported below API 17.	Set exported="false" explicitly.
RemoteControlClient recycles bitmaps on its own starting from API 19.	Check the build version.

C. *App Resource Bugs*

App resources such as images, strings, and layouts are integral parts of an app. Developers can declare the resources directly in the source code or they can externalize resources in a separate directory. However, developers are recommended [35] to externalize app resources in a separate directory named res for maintainability. Furthermore, developers can also provide alternate resources for specific device configurations by grouping them in specially-named resource directories. The external resources are referenced in the source code or other resource files using their unique resource IDs. Our dataset has 89 crash bugs in 63 apps caused by defining or referencing resources incorrectly. The distribution of app resources crash bugs according to their origin in different types of resources is shown in Fig. 4.

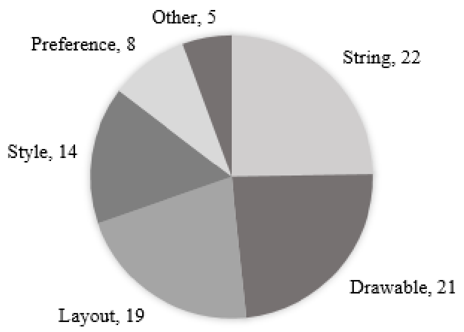

Fig. 4. Distribution of app resources crash bugs in different types of resources.

String: A string resource provides text strings for apps. It can be referenced from apps code or other resource files. Our dataset has 5 crash bugs caused by defining string resources incorrectly and 4 crash bugs caused by referencing missing string resources or referencing resources incorrectly. A string resource can be provided in different languages. However, developers should provide a default string resource so that an app can fall back to the default string if the app does not support the requested language. Our dataset has 12 crash bugs caused by localizing string resources incorrectly and 1 crash bug caused by not providing the default string resource.

Drawable: A drawable resource is a general concept for a graphic that can be drawn to the screen. It can be retrieved using APIs or applied to other XML resources such as layout using attributes. Our dataset has 10 crash bugs caused by accessing missing drawables or accessing drawables incorrectly, 3 crash bugs caused by defining vector drawables incorrectly, 3 crash bugs caused by applying drawable resources to views incorrectly, 3 crash bugs caused by corrupted or high-resolution drawables (PNG files), 1 crash bug caused by not following naming conventions while writing the name of a drawable resource, and 1 crash bug caused by using incorrect resource name while retrieving dynamic resource ID.

Layout: Layout resources placed in the res/layout/ directory define the architecture for the UI in activities or other UI components. The layout resources are defined in an XML file that mainly contains view elements and their containers. Our dataset has 4 crash bugs caused by defining resource files incorrectly, 4 crash bugs caused by accessing views that have not been defined in the resource files, 4 crash bugs caused by using incorrect attributes or their values to define views or containers, 3 crash bugs caused by referencing resources with incorrect resource IDs, 2 crash bugs caused by defining the same ID for different views or containers, 1 crash bug caused by using a RippleView that does not support older versions, and 1 crash bug caused by using an incorrect container.

Style and Colors: A style is a collection of attributes that specify the look and format for a view or window. A style applied to an entire activity or app instead of an individual view is called a theme. Our dataset has 10 crash bugs in 9 apps caused by setting styles incorrectly or using incompatible libraries for setting themes. Moreover, our dataset has 4 crash bugs in 4 apps caused by setting or retrieving colors incorrectly.

Preference and Other: Developers can provide a setting screen for their apps, which is built using various subclasses of the Preference class declared in an XML file. Each subclass provides its own specialized properties and user interface. Our dataset has 8 crash bugs in 8 apps caused by incorrectly defining preferences in XML files. Our dataset has other 5 crash bugs in 5 apps that were fixed by setting proguard rules.

App developers should 1) check references of the missing resources, 2) use appropriate resource files and naming conventions when defining resources, and 3) check string formatting when localizing string resources.

D. *Permission-based Security Bugs*

Android provides a permission-based security [36] to protect the system and apps resources. The permission-based security is implemented in the Android manifest file by defining permissions via <permission> elements and protecting resources via the android:permission attribute. Sensitive system resources such as contacts are protected with the system-defined permissions, whereas sensitive app resources such as exported components are protected by defining custom permissions. Apps willing to access the protected system or app resources must declare the permissions via <uses-permission> or <uses-permission-sdk-23> elements. The declared permissions are granted by users during install-time or runtime depending on the target Android version of the apps. Fig. 5 shows the number of crash bugs caused by the incorrect use of custom, system, and runtime permissions. Out of 59 permission-based security crash bugs, 32 crash bugs are caused by the runtime permission that has been introduced in the API level 23.

In the dataset, 21 apps access permission-protected system resources without declaring the respective system permissions, resulting in 22 crash bugs with the *SecurityException*. Developers have used various techniques to fix the crash bugs. For example, they have declared the missing permissions in the Android manifest file, caught the exception and showed an error message, avoided the use of the sensitive resources that required permissions, and performed a null check on resources that required permission in some Android versions.

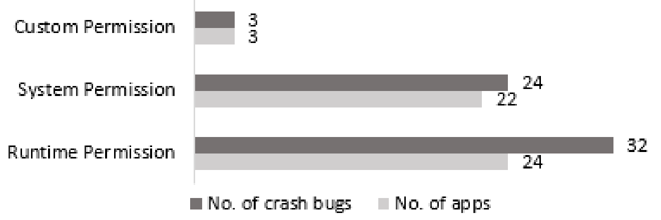

Fig. 5. Types of permission-based security crash bugs.

In Android, developers can temporarily grant permissions to read and write protected content URIs by using the flags *FLAG_GRANT_READ_URI_PERMISSION* and *FLAG_GRANT_WRITE_URI_PERMISSION*, respectively. However, the permissions to read and write the URIs do not persist permanently, which may cause an app to crash. The dataset has 2 crash bugs caused by accessing protected content URIs while the temporarily granted permissions are no longer available. The crash bugs were fixed by setting the flag *FLAG_GRANT_PERSISTABLE_URI_PERMISSION*.

In addition to the system permissions, developers can define custom permissions to protect the sensitive resources created by developers. Apps willing to access the protected app resources of another app must declare the custom permissions in their Android manifest files. In our dataset, one app protects an exported component with a custom permission without defining the permission and another app accesses a protected component without declaring the custom permission, resulting in 2 crash bugs. One serious drawback of the custom permission is that the app defining a custom permission must be installed prior to the app declaring the custom permission. Otherwise, a *SecurityException* is thrown. Our dataset has one app that crashes due to the apps install order.

The resources protected with the system or custom permissions cannot be accessed by apps unless the permissions are granted by users during install-time or runtime depending on the target Android version. Since Android 6.0 (API level 23), users have to grant the permissions during runtime. However, developers must implement the runtime permission [37] in the source code. Failure to implement the runtime permission may result in crash bugs with the *SecurityException*. In our dataset, developers have not implemented the runtime permission in 20 apps while targeting API level 23 or greater, resulting in 25 crash bugs. They have fixed 23 crash bugs by implementing the runtime permission, 1 crash bug by forcing the app to run on the API level 22 or below, and the remaining 1 crash bug by updating a library that implements the runtime permission. Our dataset has also 3 crash bugs in 3 apps caused by the incorrect implementation of the runtime permission.

Since API level 23, users can either deny permissions during runtime or revoke permissions any time. Therefore, developers should prohibit the users from accessing the protected resources when the permissions are denied or revoked. In addition, they should show an informative error message to the users. In our dataset, developers have not prohibited users from accessing the protected resources when the permissions were denied or revoked, resulting in 4 crash bugs. They have fixed 3 crash bugs by prohibiting the users from accessing the resources and the remaining one crash bug by catching the exception.

App developers should 1) ensure that the system permissions are declared for each protected resource used in the app, 2) implement the runtime permission in the apps targeting API level 23 or above, and 3) prohibit users from accessing the protected resources when the permissions are denied or revoked by them.

E. Device Orientation Bugs

Mobile devices can be rotated to change the screen orientation, which needs to be reflected in the apps running on the devices. When a device is rotated, the Android system automatically reloads the running app with alternate resources that match the new device configuration by restarting (destroying and recreating) the running activity of the app. Furthermore, developers can retain or recreate a fragment instance when an activity is restarted during device orientation. The process may create a range of problems if the activity and fragment state transitions are not handled appropriately. Our dataset has 58 crash bugs in 48 apps caused by incorrectly handling apps behavior during device orientation.

Developers have used various techniques to fix the crash bugs. They have fixed 44 crash bugs by correctly handling activity state transitions, correctly retaining or recreating fragments instances during activity restart, checking null on various objects, retrieving resources appropriately, and catching exceptions. Developers can prevent the restart of the running activity by declaring the android:configChanges attribute with the value "orientation" in the <activity> element of the Android manifest file, which they have used to fix 10 crash bugs. Furthermore, if the running activity is prevented from the restart by setting the android:configChanges attribute and the device is rotated, the system calls the *onConfigurationChanged()* method, which developers can implement to handle the configuration change. Our dataset has 1 crash bug caused by incorrectly handling new configuration in the *onConfigurationChanged()* method. Moreover, Android allows developers to use a hard value for screen orientation such as portrait and landscape to fix the orientation. Developers have used the hard values in the android:screenOrientation attribute to fix 3 crash bugs.

App developers should 1) use appropriate callback methods of activities and fragments in the appropriate order, 2) prevent restart of the activities during device orientation if the new configuration does not need to be reflected in the apps, and 3) use a hard value for the orientation if the app is not designed for different orientations.

F. Inter-component Communication Bugs

Except content providers, components of an app can communicate with each other via an intent, which is an abstract definition of an operation to be performed. An intent can be explicit or implicit based on whether it specifies a target component name. In addition to inter-component communication within an app, a component of an app can also communicate with a component of another app via intents. Inter-app communication can be performed only if the target component is exported. Our dataset has 49 crash bugs in 40 apps caused by the incorrect implementation of inter-component communication within an app and among apps. The types of inter-component communication crash bugs in the dataset are shown in Fig. 6.

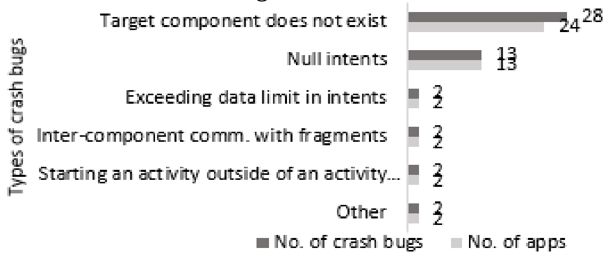

Fig. 6. Types of inter-component communication bugs.

An activity component communicates with another activity component using one of the various forms of *startActivity(Intent)* method. It can start an activity component or a data URL declared in the intent, which throws an *ActivityNotFoundException* if the activity or the URL does not exist to run the given intent. Our dataset has 28 crash bugs in 24 apps that throw the *ActivityNotFoundException*. Developers have used various techniques to fix the bugs. They have fixed 11 crash bugs by catching the exception and showing an error message, one crash bug by catching the exception and returning a fake intent to *onActivityResult()*, 2 crash bugs by catching the exception and showing a custom URL, and 2 crash bugs by correcting the URLs passed to the intents. Developers have also used techniques that verify the existence of activities or URLs before calling the *startActivity(Intent)* method. Android allows developers to query the *PackageManager* to verify the existence of an activity, which has been used to fix 6 crash bugs. One crash bug was fixed by checking the existence of a contact URI. Developers have used some other techniques to fix 5 crash bugs such as removing the *startActivity(Intent)* method, using fully qualified class name of the activity in the Android manifest file, recreating the activity, removing the parallel request, and inverting the native activity preferences.

The target component that receives the intent during inter-component communication can utilize various attributes of the intent. However, developers should validate the attributes of the received intent before utilizing them. In the dataset, developers have not validated the received intents, resulting in 13 crash bugs with the *NullPointerException*. The bugs were fixed by checking the null condition.

In addition to the 28 *ActivityNotFoundException* crash bugs and 13 *NullPointerException* crash bugs, our dataset has

8 other crash bugs in 8 apps. The *startActivity(Intent)* method can be called only from an activity context, unless the intent has *FLAG_ACTIVITY_NEW_TASK* flag. In our dataset, developers have called the method from outside of an activity context without setting the flag, resulting in 2 crash bugs. In Android, an activity can start another activity using *startActivityForResult()* or *startIntentSenderForResult()* methods to get back a result. However, if a fragment starts an activity to get back a result, the result is returned back to its parent activity. Our dataset has 2 crash bugs caused by incorrectly receiving the results in the fragments. Developers have fixed one crash bug by returning the result to the parent activity and one crash bug by calling *onActivityResult()* with the super keyword in the fragment. Android imposes a limitation on the amount of data transferred through an intent, which is currently 1 Mb. Our dataset has 2 crash bugs caused by transferring more than 1 Mb of data through intents. Our dataset has also one app that performs inter-component communication through an implicit intent containing a bundle with a custom Parcelable class. However, the intent gets intercepted by another app that attempts to unparcel the bundle, resulting in one crash bug with *ClassNotFoundException*. Finally, one app in the dataset attempts to perform inter-app communication without exporting the component, resulting in one crash bug.

App developers should 1) query the *PackageManager* to check the availability of the target component before calling the *startActivity(Intent)* method and 2) check null condition on an intent before performing any operations on the intent.

G. *Memory Bugs*

Android apps run on diverse categories of memory constraint devices. Although memory size of mobile devices is constantly increasing, users are also increasingly relying on mobile apps to perform memory intensive tasks, which may result in memory leak leading to crash if the apps are not optimized for memory usage. Our dataset has 35 apps that abuse memory, resulting in 49 crash bugs.

Developers have used various techniques to fix the crash bugs. Android sets a hard limit on the heap size allotted for each app. Developers have enabled the large heap by setting the android:largeHeap attribute in the Android manifest file to fix 3 crash bugs. However, the technique does not guarantee an increase in the available memory if the device is constrained by the total available memory. Among 12 bitmap related memory bugs in 10 apps, developers have fixed 7 crash bugs by downscaling images, 2 crash bugs by downscaling and recycling images, 1 crash bug by converting images to low-quality format (PNG to WEBP), and 2 crash bugs by discarding a library or using another image library. Developers have also fixed 12 crash bugs in 8 apps by reducing memory usage through different techniques such as writing data directly to a disk, clearing data periodically, detecting low memory situation and preventing further memory usage, preventing parallel processing of multiple events, and loading data in batches. Developers have used various other techniques to fix 16 memory bugs in 15 apps such as changing UI design, releasing UI resources, reusing views, preventing infinite dialog, releasing references or using weak references, handling files properly, refactoring source code, and preventing access to cache for multiple requests. Furthermore, developers have fixed 4 crash bugs by catching exceptions and 1 crash bug by catching the exception and decreasing maximum cache size. Android can also claim memory by killing an app. Our dataset has one crash bug caused by the side effect of the memory claim process. The bug was fixed by staring a service component in the foreground using the *startForeground()* method.

App developers should 1) downscale images before using in the app, 2) perform recycling on the images if the app targets API level 10 or below, and 3) release references or use weak references.

H. *Device and OS Bugs*

Android apps run on devices with various device configurations. Developers need to consider different device configurations while developing apps because certain features of apps may not be supported by some devices. Our dataset has 3 crash bugs in 2 apps caused by performing telephony services on the devices such as tablets that do not support the operation, 1 crash bug caused by unavailability of the external storage in a device, 1 crash bug caused by unavailability of the vibrator in a device, and 1 crash bug caused by handing context menu incorrectly in tablets.

Our dataset has 4 crash bugs in 4 apps caused by the bugs in the Android OS. Furthermore, device vendors customize the Android OS for various purposes. Our dataset has 8 crash bugs in 7 apps caused by handling the customized OS or customized stock app features incorrectly in Samsung and HTC devices. Our dataset has also 1 crash bug caused by unavailability of the Android System WebView, which has been detached from the OS since API level 21 and 1 crash bug caused by unavailability of a native library in a device.

App developers should 1) check whether the device has the requested feature such as telephony before accessing it and 2) use caution when interacting with the OS or stock apps through APIs, especially for Samsung devices.

I. *Processes and Threads Bugs*

Each app in Android runs in a separate process. Furthermore, all the components of an app run in the same process by default. However, developers can specify a component to run in another process by declaring android:process attribute in the Android manifest file. The components of the same app running on different processes have to follow inter-process communication (IPC) rules. Our dataset has 2 crash bugs in 2 apps caused by the incorrect implementation of the IPC for the components running on different processes. Developers have fixed the bugs by changing the processes in the Android manifest file.

When an app is launched, the system creates a thread of execution for the app, usually called the main or UI thread. Performing long-running operations on the main thread may block the thread, eventually leading to a crash. For example, an app in our dataset runs its media operation on the main thread, resulting in a crash bug. Furthermore, Android does not allow network operations to be performed on the main thread starting from Android 3.0. However, the behavior can be overridden by setting a thread policy, which is a bad practice. Our dataset has 6 apps that perform the network operations on the main thread, resulting in 6 crash bugs with the *NetworkOnMainThreadException*. Developers have fixed 5 crash bugs by performing the network operations on the AsyncTask or background threads. The remaining one crash bug was fixed by setting a thread policy. In contrast to performing long-running operations, UI operations must be performed on the main thread. Our dataset has 9 apps that do not perform the UI operations on the main thread, resulting in 9 crash bugs with two types of exceptions: *CalledFromWrongThreadException* and *RuntimeException* with the message "can't create handler inside thread that has not called Looper.prepare()". Developers have fixed 4 crash bugs by performing the UI operations on the main thread, 3 crash bugs by defining a handler on the main thread, 1 crash bug by using the AsyncTask, and 1 crash bug by removing the UI operation. Android imposes a limitation on the maximum pool size of threads, which is 128. Our dataset has one app that crosses the limitation, resulting in one crash bug with the *RejectedExecutionException*. The bug was fixed by using a handler.

App developers should 1) follow IPC rules for the components of an app with different processes, 2) not perform long-running tasks and network operations on the main thread, and 3) perform UI operations on the main thread.

J. Back and Up Navigation Bugs

Consistent navigation is an essential element for overall user experience [38]. The Up and Back buttons play a key role in app navigation. The Up button is used to navigate within an app based on the hierarchical relationships between screens, whereas the Back button is used to navigate through the history of screens in reverse chronological order. Our dataset has 17 crash bugs in 16 apps that are either caused by the incorrect implementation of the back and up navigation or manifested due to the back navigation.

Back and up navigation is implemented in two steps: 1) a logical parent activity is specified in the Android manifest file and 2) NavUtils APIs are used to synthesize a new back stack. Our dataset has 3 apps that do not specify or incorrectly specify the logical parent activities in their Android manifest files, resulting in 3 crash bugs. Our dataset has also 2 apps that do not implement the NavUtils APIs, resulting in 2 crash bugs. Furthermore, the home screen in apps should not offer the up navigation. However, an app in the dataset offers the up navigation in the home screen, resulting in one crash bug. Moreover, when the back button is pressed, the system calls the *onBackPressed()* method. Its default implementation simply finishes the activity. However, developers can customize the implementation. An app in the dataset manages fragment's back stack in the *onBackPressed()* method without checking the empty back stack, resulting in one crash bug.

In addition to the back and up implementation related crash bugs, our dataset has 10 crash bugs in 9 apps that manifested due to the back navigation. In all the cases, the tasks performed by the apps were interrupted by back events, resulting in crash bugs. Developers have fixed the bugs by checking null on various objects, using contexts with appropriate scope, catching the exception, checking *isFinishing()* on an activity, and handling fragments' state transitions appropriately.

App developers should check the tasks that can be interrupted by the back event leading to the *NullPointerException* and perform a null check on the objects appropriately.

V. DISCUSSION

The first goal of this empirical study is to understand the distribution or prevalence of Android-specific crash bugs in Android apps. The empirical results show that 36% (672 out of 1,862) of the crash bugs are Android-specific, which is significant number given that the Android apps are mostly written in the Java programming language. Therefore, app developers need to consider various types of Android-specific bugs while testing their Android apps.

The second goal of this empirical study is to understand the nature of Android-specific crash bugs and their root causes. The Android-specific crash bugs identified in this study have been placed in ten different categories based on their distinctive properties. The app state and UI related crash bugs are clearly on the top of the list. Given the UI-centric nature of Android apps and their components with distinct states, the result is not surprising. Most of the bugs in this category are caused by implementing state transitions incorrectly, particularly during adverse conditions or system events such as an activity destruction or restart. The app state and UI related crash bugs also manifested during the back and up navigation and device orientation. While some of the crash bugs in these categories are caused by the incorrect implementation of back and up navigation and device orientation, most of the crash bugs manifested due to the adverse conditions such as interrupting tasks by pressing the back button and rotating devices. Most of the existing UI testing techniques [8, 39, 40, 41, 42] cannot detect the bugs because they do not handle the adverse conditions. However, some testing techniques [43, 44, 45, 46] have been proposed that consider some of the adverse conditions or system events. The second in the list of the highest number of Android-specific crash bugs is the compatibility bugs. Most of the compatibility crash bugs are caused by using incompatible APIs or dependencies containing incompatible APIs. Given that the Android apps are highly dependent on APIs provided by the platform [47, 48], developers need to frequently update the APIs and support libraries in the released apps. However, McDonnell et al. [49] found that developers are slow in adopting new APIs or changes in existing APIs. It has also been established through an empirical study [50] that unstable or fault-proneness APIs impact the success of Android apps. Therefore, developers need to use stable APIs if possible. Moreover, developers can use tools [51, 52] for detecting API compatibility issues in Android apps.

Permission-based security, inter-component communication, and app resources are other major categories with the Android-specific crash bugs. The permission-based security crash bugs are mostly caused by failures to declare the required system permissions and failures to implement the runtime permission while targeting API level 23 or greater. Researchers have proposed various tools [53, 54, 55] that map sensitive APIs to their permissions. However, there is a lack of tools integrated with the development environment that warn developers about the required permissions based on the sensitive APIs used in the apps. Most of the inter-component communication crash bugs are caused by failures to handle the exceptional conditions such as unavailability of the target components to run the given intents and extracting data from null intents. In addition to UI and stress testing techniques such as monkey [56], researchers have proposed various tools and techniques [57, 58, 59, 60, 61] to analyze and test inter-component communication, which can be useful for developers in detecting the crash bugs. The app resources crash bugs are mostly caused by retrieving resources incorrectly or retrieving missing resources. In addition, several crash bugs are also caused by defining resources such as layout, string, drawable, and style incorrectly in XML files. The app resources crash bugs can be detected through UI testing techniques. However, there is hardly any tools available that specifically target app resources bugs.

The final goal of this empirical study is to understand the various solutions implemented by developers in fixing the Android-specific crash bugs. Each category of the Android-specific crash bugs requires specific solutions to fix the bugs, as discussed in Section IV. Therefore, it is difficult to generalize the solutions implemented by developers in fixing the crash bugs. However, there are some general patterns in fixing the crash bugs. In addition to the specific solutions implemented by developers in fixing the specific crash bugs, developers have fixed 56 Android-specific crash bugs, which is 8% of the total Android-specific crash bugs, by catching the exceptions. In contrast to our result, a study performed by Zhang et al. [62] on 8 popular Android apps shows that 22% of the bugs were fixed by adding, modifying, or refactoring exception handlers. However, their study included Java-related bugs. One of the acute problems in the Java programming or the object-oriented programming is null-pointer dereferencing [63]. Android inherited the same problem. Developers have fixed 55 Android-specific crash bugs by checking the null condition. In contrast to our result, Coelho et al. [64] found that 27.71% of the exception stack traces in the bug reports of Android apps contained a *NullPointerException*. However, their study included Java-related bugs.

VI. 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 672 Android-specific crash bugs from 418 free and open source apps. Although the dataset is fairly large for this kind of studies, the results of this study may not be generalized in a larger context. Furthermore, we have performed the study on free and open source Android apps. Therefore, the results may not be generalized for all kinds of apps.

Threats to internal validity relate to the methodology used to perform the study. Our study depends heavily on manual analysis, which has various limitations that may influence the results. We identified crash bugs by searching issue trackers with three different keywords: crash, force close, and exception. Although highly unlikely, it is possible that the keywords might not have been used in the crash bug reports, resulting in the exclusion of the crash bugs in our dataset. Furthermore, we manually performed the analysis to identify the Android-specific crash bugs, their root causes, and solutions, which may influence the results. However, the results were cross-validated to mitigate the threat. Moreover, we have provided all the data used in this study for interested readers [30].

VII. CONCLUSION

In this paper, we performed an empirical study to understand the Android-specific crash bugs, their prevalence, root causes, and solutions by analyzing 1,862 confirmed crash reports of 418 open source Android apps. The empirical results show that Android apps have a significant number of (672 out of 1,862) Android-specific crash bugs that belong to ten different categories based on their distinctive properties. Although the Android-specific crash bugs in each category have specific root causes, the empirical results show that they mainly originate due to failure to handle special or exceptional cases, failure to update apps promptly when the changes are made in the platform, and failure to handle adverse conditions. Therefore, to prevent the crash bugs, in addition to handling exceptional cases and adverse conditions during app development, developers need to update apps frequently reflecting changes made in the platform.

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

[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, no. 1 (2015): 40-64.

[3] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 429-440. IEEE, 2015.

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

[5] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp. 1-10. IEEE, 2015.

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

[7] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. D. Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. Enabling mutation testing for Android apps. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 233-244. ACM, 2017.

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

[9] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su. Large-Scale Analysis of Framework-Specific Exceptions in Android Apps. In Proceedings of the 40th International Conference on Software Engineering, pp. 408-419. ACM, 2018.

[10] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru. An empirical analysis of bug reports and bug fixing in open source android apps. In Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 133-143. IEEE, 2013.

[11] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile oses: A case study with android and symbian. In 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE), pp. 249-258. IEEE, 2010.

[12] M. E. Joorabchi, M. Mirzaaghaei, and A. Mesbah. Works for me! characterizing non-reproducible bug reports. In Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 62-71. ACM, 2014.

[13] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In Proceedings of the 10th international conference on Mobile systems, applications, and services, pp. 267-280. ACM, 2012.

[14] Y. Liu, C. Xu, S. Cheung, and V. Terragni. Understanding and detecting wake lock misuses for android applications. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 396-409. ACM, 2016.

[15] Y. Liu, C. Xu, and S. Cheung. Characterizing and detecting performance bugs for smartphone applications. In Proceedings of the 36th International Conference on Software Engineering, pp. 1013-1024. ACM, 2014.

[16] H. Shahriar, S. North, and E. Mawangi. Testing of memory leak in Android applications. In 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering (HASE), pp. 176-183. IEEE, 2014.

[17] A. K. Jha, S. Lee, and W. J. Lee. Developer mistakes in writing Android manifests: an empirical study of configuration errors. In Proceedings of the 14th International Conference on Mining Software Repositories, pp. 25-36. IEEE Press, 2017.

[18] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In Proceedings of the Ninth European Conference on Computer Systems, p. 18. ACM, 2014.

[19] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 588-598. ACM, 2014.

[20] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 94-105. ACM, 2016.

[21] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing-and touch-sensitive record and replay for android. In Proceedings of the 35th International Conference on Software Engineering (ICSE), pp. 72-81. IEEE, 2013.

[22] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for android. In ACM SIGPLAN Notices, vol. 50, no. 10, pp. 349-366. ACM, 2015.

[23] Z. Qin, Y. Tang, E. Novak, and Q. Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In Proceedings of the 38th International Conference on Software Engineering, pp. 571-582. ACM, 2016.

[24] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In MOBILESoft'16, pp. 88-99. IEEE, 2016.

[25] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Auto-completing bug reports for android applications. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 673-686. ACM, 2015.

[26] A. K. Jha and W. J. Lee. Capture and Replay Technique for Reproducing Crash in Android Applications. In Proceedings of the 12th IASTED International Conference in Software Engineering, pp. 783-790. 2013.

[27] A. K. Jha, S. Jeong, and W. J. Lee. Value-deterministic search-based replay for android multithreaded applications. In Proceedings of the 2013 Research in Adaptive and Convergent Systems, pp. 381-386. ACM, 2013.

[28] S.H. Tan, Z. Dong, X. Gao, and A. Roychoudhury. Repairing Crashes in Android Apps. In Proceedings of the 40th International Conference on Software Engineering, pp. 408-419. ACM, 2018.

[29] F-Droid, Free and Open Source Android App Repository - https://f-droid.org/

[30] Dataset - Android-specific crash bugs. https://github.com/HiFromAjay/Android-specific-Crash-Bugs/blob/master/Dataset.pdf

[31] C.B. Seaman. Qualitative methods in empirical studies of software engineering. IEEE Transactions on software engineering 4 (1999): 557-572.

[32] M.B. Miles, A.M. Huberman, and J. Saldaña J. Qualitative Data Analysis: A Methods Sourcebook (3rd ed.). SAGE Publications, Inc. 2013.

[33] Context - https://developer.android.com/reference/android/content/Context.html

[34] A. K. Jha, and W. J. Lee. An empirical study of collaborative model and its security risk in Android. Journal of Systems and Software. 137C (2018) pp. 550-562

[35] Resources Overview - https://developer.android.com/guide/topics/resources/overview.html.

[36] A. K. Jha and W. J. Lee. Analysis of Permission-based Security in Android through Policy Expert, Developer, and End User Perspectives. J. UCS 22, no. 4 (2016): 459-474.

[37] Requesting permissions at run time - https://developer.android.com/training/permissions/requesting.html

[38] Navigation with Back and Up - https://developer.android.com/design/patterns/navigation.html

[39] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 258-261. ACM, 2012.

[40] W. Yang, M. R. Prasad, and T. Xie. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In FASE, vol. 13, pp. 250-265. 2013.

[41] T. Azim, and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In Acm Sigplan Notices, vol. 48, no. 10, pp. 641-660. ACM, 2013.

[42] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In Acm Sigplan Notices, vol. 48, no. 10, pp. 623-640. ACM, 2013.

[43] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In Proceedings of the Seventh International Conference on Software Testing, Verification and Validation (ICST), pp. 183-192. IEEE, 2014.

[44] C. Q. Adamsen, G. Mezzetti, and A. Møller. 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, 2015.

[45] Z. Shan, T. Azim, and I. Neamtiu. 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, 2016.

[46] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su. Guided, stochastic model-based GUI testing of Android apps. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 245-256. ACM, 2017.

[47] M. D. Syer, B. Adams, Y. Zou, and A. E. Hassan. Exploring the development of micro-apps: A case study on the blackberry and android platforms. In 2011 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 55-64. IEEE, 2011.

[48] M. D. Syer, M. Nagappan, A. E. Hassan, and B. Adams. 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. IBM Corp., 2013.

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

[50] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. 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, 2013.

[51] L. Wei, Y. Liu, and S. Cheung. Taming android fragmentation:Characterizing and detecting compatibility issues for android apps. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 226-237. ACM, 2016.

[52] L. Li, T. F. Bissyandé, H. Wang, and J. Klein. 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, 2018.

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

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

[55] 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, no. 6 (2014): 617-632.

[56] UI/Application Exerciser Monkey - https://developer.android.com/studio/test/monkey.html

[57] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer. An empirical study of the robustness of inter-component communication in Android. In 2012 42nd annual IEEE/IFIP international conference on Dependable systems and networks (DSN), pp. 1-12. IEEE, 2012.

[58] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In Proceedings of the 37th International Conference on Software Engineering, pp. 77-88. IEEE Press, 2015.

[59] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In Proceedings of the 37th International Conference on Software Engineering, pp. 280-291. IEEE Press, 2015.

[60] A. K. Jha, S. Lee, and W. J. Lee. Modeling and test case generation of inter-component communication in Android. In Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems, pp. 113-116. IEEE Press, 2015.

[61] R. Sasnauskas, and J. Regehr. Intent fuzzer: crafting intents of death. In Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), pp. 1-5. ACM, 2014.

[62] P. Zhang, and S. Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. ACM Transactions on Software Engineering and Methodology (TOSEM) 23, no. 4 (2014): 32.

[63] B. Meyer. Ending null pointer crashes. Communications of the ACM 60, no. 5 (2017): 8-9.

[64] R. Coelho, L. Almeida, G. Gousios, and A. V. Deursen. Unveiling exception handling bug hazards in Android based on GitHub and Google code issues. In 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR), pp. 134-145. IEEE, 2015.