Original Research Paper

A Static Analysis of Android Source Code for Lifecycle Development Usage Patterns

[1]Noura Hoshieah, [2]Samer Zein, [3]Norsaremah Salleh and [4]John Grundy

[1]Software Engineering Master Program, Birzeit University, Birzeit, Palestine
[2]Department of Computer Science, Birzeit University, Birzeit, Palestine
[3]Department of Computer Science, International Islamic University, Kuala Lumpur, Malaysia
[4]Faculty of Information Technology, Monash University, Clayton, Australia

Article history
Received: 26-11-2018
Revised: 28-12-2018
Accepted: 05-01-2019

Corresponding author:
Noura Hoshieah
Software Engineering Master Program, Birzeit University, Birzeit, Palestine
Email: noura.hoshieah.1@gmail.com

Abstract: Building robust Android apps is a non-trivial task that requires skilled developers to understand various Android platform peculiarities. However, among the Android developers community, a large fractions are considered to be novice and inexperienced developers. One of the main peculiarities in the Android app development is the activity lifecycle model. A developer needs to have deep understanding of the different lifecycle states and callback methods that an Android activity can go through during its runtime. These callback methods are called by the system whenever an app activity changes its state. The developer needs to override appropriate callback methods correctly to avoid app memory leaks and data loss or other phone resource compromise. Detailed static analysis of software applications provides actionable insights and helps us to understand how applications are actually built. Although there have been many studies focusing on static analysis of Android apps in the areas of testing, quality, design, privacy and security; no studies to date focus on lifecycle development practices and usage patterns thus far. In this paper, we analyzed 842 open-source Android apps containing 5577 activities to explore and understand how Android developers actually comply with best practices regarding the Android activity lifecycle model. We developed a tool named SAALC that is capable of analyzing Android activities and extracting valuable information about lifecycle callback methods usage. Our results show, which callback methods are implemented and the nature of the code they contain. The results also show incorrect implementation of the callback methods and incorrect acquiring and releasing of system resources in many Android apps and we argue that a relatively large fraction of Android developers do not sufficiently well understand the app lifecycle model. We also discuss our results in comparison to the Android app lifecycle model best practices.

Keywords: Android, Activity Lifecycle, Static Analysis, Application, Mobile Apps

Introduction

Mobile applications (apps) usage has increased exponentially with millions of apps being available at the online stores (Wasserman, 2010). Nowadays, users rely on mobile apps to deliver their daily tasks. Indeed, mobile apps cover various fields such as social, business, health, productivity and gaming to mention a few (Dehlinger and Dixon, 2011). Moreover, mobile devices offer the same functionality as PC through wireless, web browsers, video and audio. At the same time, the mobile app development is not a trivial task and has its own challenges (Zein *et al*., 2017).

Android is the most common mobile platform in use and the most popular mobile open source Operating Systems (OS) in the mobile apps market (Lamba *et al*., 2015). Industrial analysts expect that the Android platform will remain the dominant mobile vendor for many years to come. Google Play is the main online store providing Android apps. Since the

Android first release in 2008, developers have been heavily contributing in developing new apps that facilitate various user needs. As a result, In April 2017, the number of available Android apps has exceeded 2.8 millions (Wasserman, 2010; Number of apps, n.d.). Further, the number of worldwide downloaded Android apps from Google play was estimated in billions in 2016 to 2017 (Number of apps, n.d). Accordingly, the complexities of mobile apps increase to fulfill a variety of functionality and features.

Mobile app development is different than other traditional web and desktop paradigms. Developers face a new set of challenges, including developing apps for different platforms (iOS and Android), handling the issues of OS and hardware fragmentation and managing app lifecycle conformance (Joorabchi *et al*., 2013; Zein *et al*., 2016; Franke *et al*., 2011; 2012). Even though much research has been directed to address some of these challenges, little research has been done in the area of lifecycle conformance.

When developing for Android, activities represent the User Interface (UI) and each activity goes through different states during its lifecycle. These states are running, paused, stopped and shutdown. Each activity makes transitions between these states due to some events, such as receiving an incoming call, by calling a specific callback method (Joorabchi *et al*., 2013). Android developers need to have good understanding of the lifecycle model in order to develop apps that function correctly (Franke *et al*., 2011; 2012; Zein *et al*., 2017). Google documentation provides narrative information about the lifecycle model to assist developers in building robust apps (Franke *et al*., 2011). However, a large fraction of Android developers are known to be novices and amateurs who may not properly understand or follow the lifecycle model and will end up with unreliable and faulty apps (Zein *et al*., 2017). Additionally, there are as yet no automated testing tools available for Android that enable developers to fully check the correctness of the app adherence to the lifecycle model (Zein *et al*., 2017).

This study aims to explore how android app developers actually utilize the lifecycle callback methods. More specifically, we aim at analyzing Android open-source apps to reveal how these apps are built in terms of lifecycle callback methods and the utilization of system resources such as memory, Camera, GPS, Sensors, etc. Analyzing Android app's source code is a popular recent topic (Panichella *et al*., 2015) and provides good insights about how these apps are developed and structured (Haotian and Shu, 2013). For instance, this analysis helps increase the quality of app code and improve reliability and performance of the software (Haotian and Shu, 2013; Danphitsanuphan and Suwantada, 2012). Another example is rule mining (Khatoon *et al*., 2011). Rule mining aims to extract hidden rules from existing project in order to improve new development projects (Khatoon *et al*., 2011). Further, rule mining has been used in automated defect detection for complementing the compiler work and this is done through analyzing the source code to find the most common bugs (Panichella *et al*., 2015; Khatoon *et al*., 2011). Indeed, analyzing the source code gives more insights and helps the research community and the software industry to understand how developers actually code their apps. In other cases, it can be useful to understand the architecture of the app and consequently to reduce the development time and programming effort (Haotian and Shu, 2013; Khatoon *et al*., 2011). Other benefits of analyzing source code include identification and elimination of security vulnerabilities in software (Ramos, 2016) and providing statistical measures about the code complexity and quality, such as numbers of methods, attributes, parameters, children, line of codes, depth of inheritance, algorithm complexity, coupling and coherence, etc (Danphitsanuphan and Suwantada, 2012).

Although there are a lot of studies focusing on analyzing Android source code insights and usage patterns in different fields such as testing; quality; design; privacy and security, to our knowledge, there have been no studies to date focusing on analyzing Android source code for lifecycle adherence. To address this, we conducted a quantitative study to analyze Android source code. The main aim of our study is to explore how real Android developers develop their apps in terms of usage of lifecycle callback methods. To achieve this, we analyzed 842 open-source Android apps containing 5577 activities. These apps were downloaded from the FDriod repository. Our dataset includes different apps with varying code sizes and from different categories such as Gaming, Navigation, Internet, Multimedia, etc. We built a statistical analysis tool called SAALC which is able to analyze and extract all data related to activity lifecycle callback methods. The resulting statistics reveal the usage of callback methods and where system resources, such as camera, Bluetooth, GPS, etc., are acquired and released among other important information. Also, we analyzed the nature of code implemented inside these callback methods to understand for what they are used.

More specifically, the results show that the onCreate() callback method is the one that is mostly utilized (92%) among all activities. On the other hand, the onRestart() and onStart() callback methods are about 1% and 6% respectively. Further, the onDestroy() and onStop() have 14% and 6% usage respectively (Number of apps,n.d.). We also found that the average percentages of wrongly acquired and wrongly released system resources are about 20% and 8% respectively. Such results enable us to better understand more how Android developers utilize lifecycle callback methods. Our study makes the following key contributions:

- We developed a tool named SAALC for analyzing Android activity lifecycle files generated statistics about each callback method used
- We conducted the first detailed study to explore how Android developers utilize activity lifecycle callback methods
- We generated app lifecycle adherence statistics about acquiring and releasing of system resources during the lifecycle of Android apps from a large corpus of open source Android apps
- We identified where improvements or alterations are required into aid developers

The structure of the remainder of this paper is as follows. In section 1, we show a brief background about Android and activity lifecycle and the most research and related work for our study. Section 2 explains our study design which includes the research questions, data collection method and basic statistics of our collected data. Also, we show our proposed tool and algorithms which called SAALC. Section 3 present s the results of current state of Android lifecycle callback methods and system recourse usage. We discuss the results of the study in section 4.

Background and Related Work

In Android development, the "app lifecycle" can be defined as the different activity states and the transitions between them during the runtime of an Android activity (Franke *et al*., 2012). Every Android app consists of one or more activities which manage the behavior of the app. Each activity has its own lifecycle. When an activity is first run till the moment it is released and destroyed by the system, it passes through different states. Whenever an activity goes into a new state, a special callback method is called by the system. The developers respond to state transitions by overriding relevant callback methods. This event-driven behavior model is well-known to be challenging to manage when many activities are created or when activities go through complex sets of transitions.

As with other mobile platforms android cannot preserve the state of an app during lifecycle changes due to the lack of system resources (Franke *et al*., 2012). Thus the developers themselves must ensure that no data is lost when the state changes (Franke *et al*., 2011; Zein *et al*., 2017). An activity changes its state due to some event such as the mobile phone receiving an incoming call, a user interaction event, or starting another activity to mention a few. Mobile operating systems such as Android are very efficient when dealing with the device resources such as memory, CPU and battery (Franke *et al*., 2012). Thus, the Android OS may swap out or kill an activity without saving its current state in case of lack of resources (Franke *et al*., 2012). It is the developer's job to make sure the app conforms correctly to the lifecycle model (Franke *et al*., 2012).

When activity callback methods are executed, the activity state will change and the control return to the system (Franke *et al*., 2011; 2012). Developers must follow Android lifecycle model documentation offered by the Android vendor (Franke *et al*., 2012). The Google official website is the main portal for the Android developer, it contains the guidelines of the Android development model lifecycle and documentation (Zein *et al*., 2017). The Android activity lifecycle model shown in Fig. 1 from the Android Developers Guide (Zein *et al*., 2017). An activity can be at one of the following states (Franke *et al*., 2011; Zein *et al*., 2017):

- Created: When the activity is first created and initiated
- Started: Activity is ready
- Resumed: Activity is ready and in the foreground, the user can start interacting with it
- Paused: when anther activity obscured the running activity (Franke *et al*., 2011)
- Stopped: when the activity is not visible on screen and running in the background, it will stay in the memory
- Destroyed: Activity is removed from the memory and lo longer exists (Franke *et al*., 2011)

In Fig. 1, the ellipse shape represents activity state while the arrows illustrate lifecycle callback methods, which the developer should override (Franke *et al*., 2011). Note that the activities will not stay on created, started, paused states for a long time because it passes them quickly (Franke *et al*., 2011; Zein *et al*., 2017).

However, researchers have found errors commonly occurring in the transitions between the states in the official model shown at Fig. 1. For instance, the study by (Franke *et al*., 2011) reversed-engineer the lifecycle of Android activities using assertion-based test cases. The results of their study show that the onStop() and onDestroy() callback methods are not guaranteed to be called by the system when the system is very low on resources. Thus, an activity may be destroyed and removed from memory without executing the code written inside onStop() and onDestroy() callback methods.

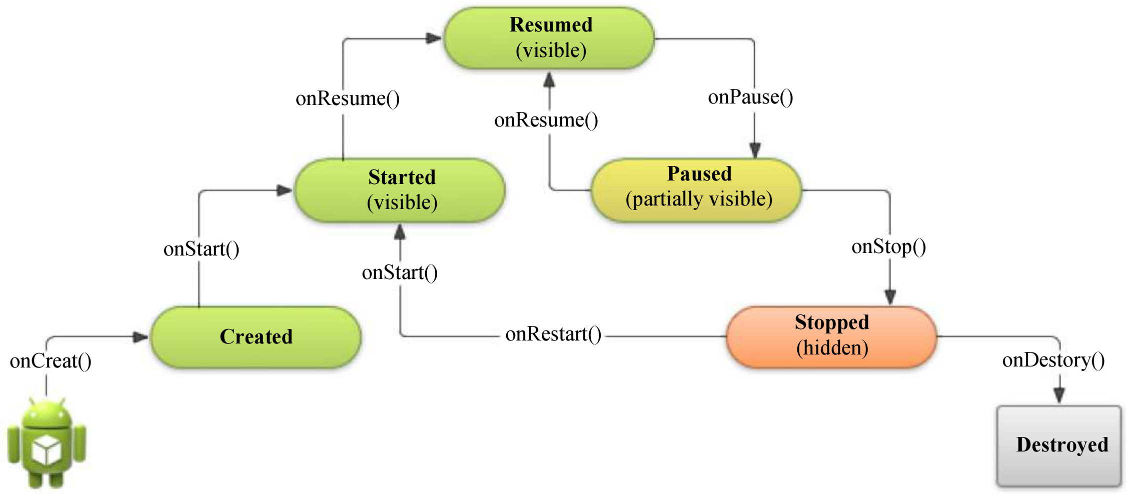

Fig. 1: Android activity lifecycle model (Franke *et al*., 2011)

Zein *et al*. (2017) developed an automated testing tool called Android Lifecycle Inspector (ALCI) to check if system resources such as camera, GPS, sensors, etc., are correctly acquired and released by developers. This tool helps novice developers to build more robust mobile apps. The approach is based on mobile software model to extract lifecycle system resources rules and create repositories for these resources. ALCI approach analyzes the source code of an application against rule models to verify that the developer has been correctly initiated and release an application system resources. Also, ALCI generates a report notification of incorrect system resource for the developer. Bartel *et al*. (2014) presented a test suite approach using taint-based static analysis called FLOWDROID for evaluating the effectiveness and accuracy of Android apps and it take in the consideration Android lifecycle and callback methods challenges. In our study, we take in consideration the issues and problems presented by (Franke *et al*., 2011).

Android static analysis has become a popular research topic. Lamba *et al*. (2015) proposed an approach of analyzing and mining 1,120 android source code applications from F-Driod using Java parser to extract API Call Usage Patterns (ACUP), methods, classes, interfaces and packages information to show the developer styles and feedback of using android platforms in the mobile application for the new developers. Batyuk *et al*. (2011) proposed a static analysis service that is able to assess the apps market and provide a user with a detailed report on the security and privacy insight level inside an app. It offers the user the ability to mitigate the malicious code and security threats inside the app. Schmidt *et al*. (2009) suggested a static analysis for Android executable which locate inside a lunix system (ELF file inside/bin) using a command readelf in order to extract the functions calls and compare them with a malware executable to detect them using three simple classifiers. Payet and Spoto (2012) used the Julia static analyzer for Java byte code inside Android apps to ensure that the apps doesn't contain programming bugs. Bartel *et al*. (2014) proposed an automatic static analyzer called SCANDAL and used it to analyze 90 Android apps in order to detect privacy leaks. Zhongyang *et al*. (2013) suggested alarm attack called DroidAlarm which used static analysis and able to parse for sensitive permissions and public interfaces for identifying potential capability leaks for Android apps. Feng *et al*. (2014) showed an approach called Apposcopy which uses a taint static analysis and represent Android code in Inter-Component Call Graph to detect control and flow of data that causes a malware. Desnos (2012) presented a static analysis algorithm which based on a customized similarity distance to decide if the developer protect them app from piracy and identify code updates and releases to find dissimilarities between versions of an app.

The above studies signify a growing interest in the software engineering research community to analyze Android source code apps. However, to our best knowledge, there are no studies so far that analyze Android apps for their lifecycle usage and conformance except one study for Zein *et al*. (2017) which used analysis for testing lifecycle conformance. However, in this study, we fill this gap by presenting the analysis of Android lifecycle activities on a large number of open source apps and describe how Android developers utilize lifecycle callback methods and the system resources acquired/released.

Study Design

We first present the key research questions investigated in this study. We also present information about our data set.

Research Questions

After analyzing 5000+ activities from various Android apps, we wanted to know how callback methods are typically utilized by developers. We developed a set of research questions to help us to understand the activity lifecycle practices commonly followed by real Android developers. First, we want to understand the usage of lifecycle callback methods in Android activities. This leads us to our first research question:

RQ1: To what extent Android developers utilize the lifecycle callback methods in developing mobile apps?

Similar to the first RQ, we are also motivated to analyze the source code related to activities and collect statistics about main system resources management, such as camera and Bluetooth, GPS, sensors, etc. We want to know whether Android developers acquired and released these system resources correctly as compared to the standard Google documentation. Thus, our second research question is:

RQ2: Do Android developers correctly acquire and release the Android system resources?

In the third RQ, we aim to analyze all activities to better understand the nature of code implemented inside the key onPause(), onStop() and onDestroy() callback methods. We decided that the nature of this code includes three categories - releasing, database or threading actions. According to the Android documentation site the database and threading actions are considered as long running code (Activity android developers, n.d.). Accordingly, this question will give us some statistics about how the developer uses these callback methods and, if they implemented a long running code approach or not:

RQ3: What is the nature of the code implemented inside onPause(), onStop() and onDestroy() callback methods? In order to answer these questions, we divided our work into four phases. Firstly, we built a statistical analysis tool which is able to read and analyze Android source code. Secondly, we generated usage statistics about the override callback methods. In the third part, we generated statistics about system resource management. In the last phase, we collected information about the nature of code inside important callback methods namely onPause(), onStop() and onDestroy(). We present our research results in section 5.

Data Collection

We collected URLs of all apps stored on the F-Droid repository and selected apps that are hosted on GitHub. F-Droid is a popular platform and online software repository which contains open source code for a very diverse range of Android apps. Each app in F-Droid is also available on Google play (FDroid, n.d.). F-Droid was selected because it provides useful categorizing and classification of the Android apps. This classification will be used to help us when analyzing Android apps according the category type of the app. As of Dec 8, 2016, there were 2001 apps in the F-Droid repository (and 1420 also on Google Play) organized into 17 categories (FDroid, n.d.). The apps were downloaded manually from their individual pages. In total, we have 842 apps in our data set from 17 categories. We manually checked the manifest XML file for each app and collected all activity files. In total, we have 5577 activities extracted from 842 Android apps as shown in Table 1.

The data shown in Table 1 indicates that the system category has the largest number of apps (Apps) which is equal to 265 in the F-droid data set; followed by multimedia (242) and Internet category (221). The largest number of activities (Activity) is found in the internet category (952) then Science and Education (624) category. Figure 2 shows a Pareto chart that shows our data set with a graphical distribution using a combination of a line and bar chart. The bar chart represents the cumulative total #Activity across each category while the line graph shows the cumulative percentage of apps. The Pareto chart reveals that 57% of activities in the data set belong to 29% of the categories inside the first five categories which are internet, science and education, multimedia, games and navigation.

SAALC Architecture and Implementation

To assist us with our static analysis work we developed a new tool called Static Analysis of Android Lifecycle (SAALC). SAALC is able to read a data set of Android activities that is written in the Java programming language. SAALC is the first tool that analyzes a dataset of Android source code to extract information related to activity lifecycle callback methods. Figure 3 shows a block diagram of the main components of SAALC as well as the approach of the analysis process.

SAALC includes the following key components:

- Java Parser component: This is an open source and free parser available at GitHub. It is used to parse and convert Android source code into AST object model (Abstract Syntax Tree). The AST object model contains a list of imported packages, methods and fields declarations for each class of a source code to mention a few. We used these declarations in the analysis of Android activities. Using an import package declaration, we can get a name of the package that declared in the class. Additionally, using a field declaration, we can get the field name and its type. Additionally, using the method declaration, we can get a method name and its body contents

- Android Source Codes Reader component: This component reads a data set of Android's activities.
- Output Report component: Produces output reports in CSV (Comma Separated Values) file format.
- Resource List component: Produces a list of resources
- Resource DB component: Resources information in an XML file
- Analyzer component: The main that applies two handling algorithms. State Analyzer: Inspects data set source code to collect statistics about callbacks methods and the natures of code inside them. Resource Analyzer: Inspects data set source code to collect statistical information about managing system's resources

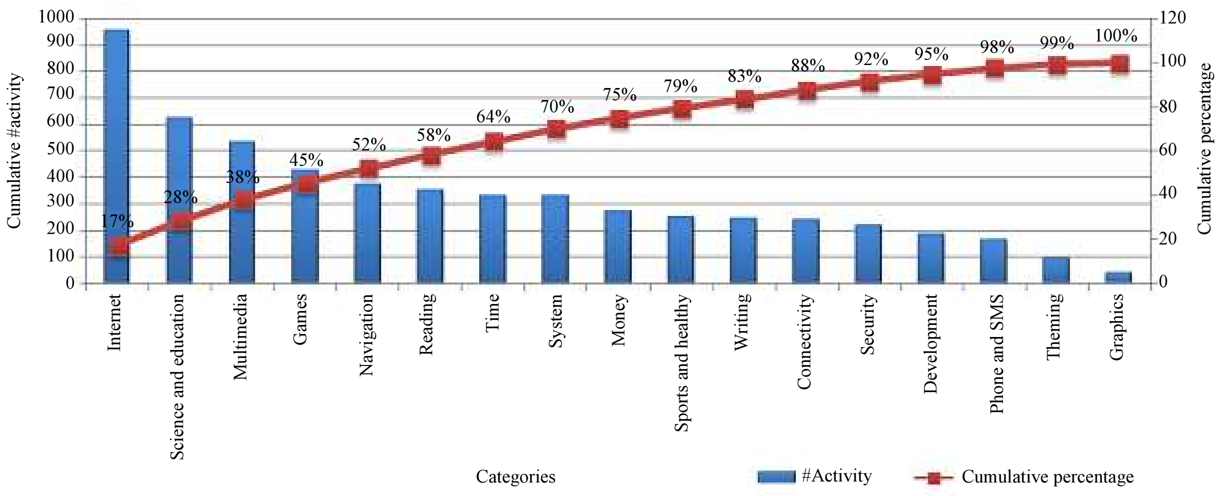

Fig. 2: Distribution of activities on data set across the 17 categories

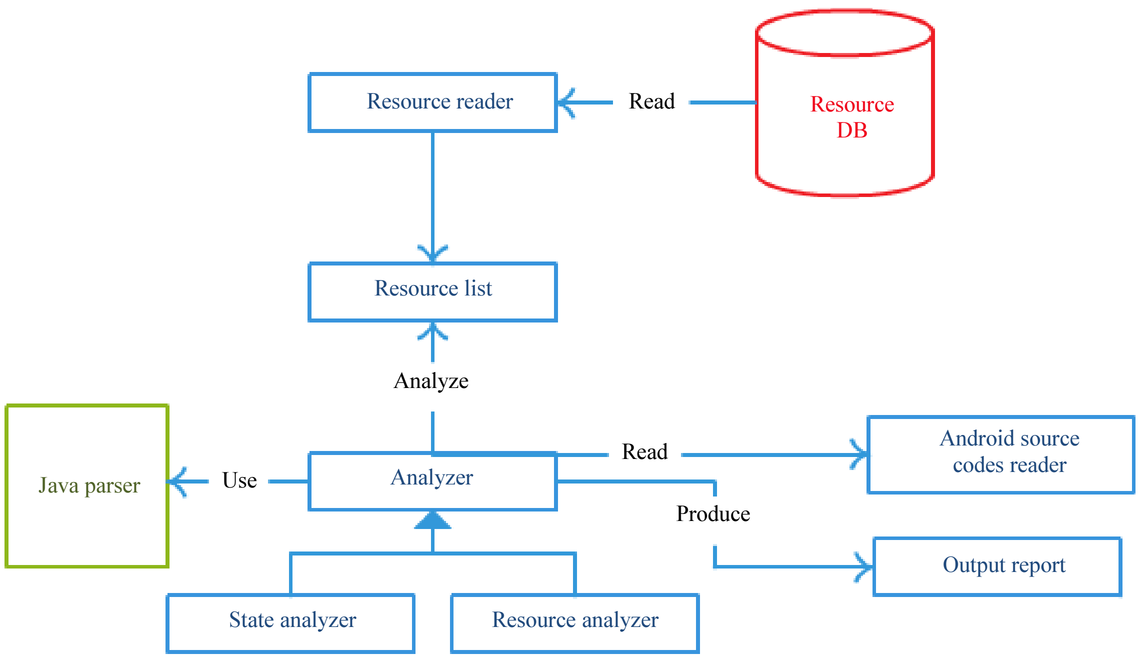

Fig. 3: Structure diagram of SAALC

Table 1: Distribution of the data set over the app categories

Category	#APP	#App Downloaded	#Activity
System	265	78	325
Multimedia	242	104	532
Games	221	93	422
Internet	217	122	952
Navigation	135	58	370
Science and Education	118	45	624
Theming	108	26	94
Reading	104	40	347
Time	104	55	328
Writing	94	37	242
Development	92	28	182
Connectivity	84	51	236
Security	68	32	215
Phone and SMS	53	15	159
Money	40	26	266
Sports and Health	35	23	247
Graphics	21	9	36
Total	2001	842	5577

The key processing steps of our approach are as follows:

- First, SAALC reads the system resources information from the repository using Read Resource component, then produces a list of resources using Resource List component
- Secondly, The Java Parser component parses Android source code and produces the detailed AST object model
- Then, using the resulting object model produced by Java Parser, SAALC applies two analysis algorithms. The State Analyzer is responsible for collecting information about each callback method such as the count of each callback method and the nature of code inside them. The Resource Analyzer inspects the source code against the system resources list
- Finally SAALC produces a results report in CSV file format using the Output Report component

We stored the system resources using the Resource DB component in an XML repository. We chose 9 system resources to track and analyze including Camera, USB, Sensor, Network, Input, GPS, Database, Bluetooth and Audio. For each of these system resources, we stored the resource name, package name, the name of acquiring and releasing methods and the name of callback methods which used to acquire and release the resource. All the above information is taken from the official Google Android site (Activity android developers, n.d.). Table 2 shows a sample repository information for Camera resource.

The analysis algorithms are able to analyze the common coding patterns applied by developers. Below are the common coding patterns is used by developers according to the study by (Zein *et al.*, 2017):

- Developer calls the acquired or release method directly inside callback method block
- Developer calls another method or nested methods inside a callback method which in turn calls the acquired or release method
- Developer calls the acquired or release method inside if, while, for, switch, try catch, threads or object block statements which are inside the callback method or other nested methods

Additionally, during our study, we found that in some cases the developers did not acquire or release system's resource inside callback methods. Instead, they manage system resources inside the event handlers. Accordingly, we analyzed Android source code against these events.

These events include methods which were overridden in the activity source code and did not a callback method. We refer to these event methods in the Results Section using the "OTHER" keyword.

The outline of our two analysis algorithms are described as follows:

State Analyzer Algorithm:

The proposed algorithm for State Analyzer can be described in pseudo-code as shown in as follows:

- **Algorithm Input:** List of all activities source code in our dataset
- **Algorithm Output:** A report result in CSV.

Algorithm basic steps:

1. Load a list of activities source code
2. For each activity in the list of activities, parse and traverse the activity source code into AST. Then, analyze the source code to find and count the occurrences of the callback methods which were used
3. If onPause(), onDestroy() or onStop() callback methods were founded in the source code, then analyze the nature of code inside each of them related to releasing, database or threading actions

Resource Analyzer Algorithm

The proposed algorithm for Resource Analyzer can be described in pseudo-code as shown in as follows:

- **Algorithm Input:** List of all activities source code, List of system's resources information
- **Algorithm Output:** Reports result in CSV

Table 2: Camera resource information

Resource name	Package name	Name of acquiring method	Name of releasing method	Name of acquire callback method	Name of release callback method
Camera	Camera	open(), startPreivew()	release(), stopPreview()	onResume()	onPause()
	Camera2	open(), startPreivew()	release(), stopPreview()		
	CameraManager	openCamera()	release()		
	CameraDevice	openCamera(), onOpened()	release(), onClosed()		

Algorithm basic steps:

1. Load a list of activities source code and list of resource information. This includes all resource names, package names, names of acquiring and releasing methods
2. For each activity in the list of activities, parse and traverse the activity source code into AST
3. For each system resources such as Camera, GPS etc, analyze the source code of each activity to find in any methods (callback or OTHER methods) where this resource has been acquired and released

Results

In this section, we present the results of our study after analyzing the data set. The results are divided into three parts as shown in subsections below. The first part is for callback methods usage counts; the second part is for system resource management; and the third for the nature of code inside some of the most important callback methods; namely, onPause(), onStop() and onDestroy().

Part I: Usage of Callback Methods

In the first stage of analysis, we counted the totals of each callback method for the 5577 activities. We present the result set in Fig. 4, which shows the number of each callback method found in the data set. Figure 4 shows the results of as a Bubble Chart. The Bubble Chart is based on three dimensions: (i) The horizontal axis represents the callback methods names;(ii) the vertical axis represents the counts of occurrence of each callback method; (iii) the bubble size indicates the third dimension which represents the cumulative percentage of callback methods. The Bubble Chart also shows that the most occurrences callback method are onCreate() (92%) followed by onResume() (23%) then onPause() (16%), onDestroy() (14%), onStart() (6%), onStop() (6%) and the last is onDestroy() (1%).

Additionally, we show the counts and percentages of callback methods in relation to the categories of the Android apps in Table 3. The highest value is (888) 16% for onCreate() callback method and category Internet followed by value (550) 10% for onCreate() and category Science Education and then the value of (514) 9% for onCreate() with category Multimedia. Moreover, The lowest values are for onRestart() callback method across all the categories. Finally, the map shows that the Internet category is the most frequently used for all callback methods.

Acquiring and Releasing of System Resources

In the second part of our analysis, we present the occurrence of each system recourse in the 5577 activities extracted from the selected apps. In order to decide on the correct/incorrect management of the system resources, we referred to the Google Android official documentation (Activity android developers, n.d.). Google Android documentation provides guidelines about system resource usage and any callback method that is responsible for acquiring or releasing the resources (Activity android developers, n.d.).

The distribution of callback methods is shown in Fig. 5 as a Column Chart. It shows the number of occurrence of each system's resource founds in the data set. We implement the analyzer over nine resources which are Camera, Audio, Bluetooth, SQLite Database, GPS, Input, Network, Sensor and USB. The results show that the total number of the resource used is (178) 3% over 5577 activities. Also, it shows that the Database occurrence equals to 52 (0.93%), so it is the most popular system's resource used by developer followed by Sensor (0.46%) then Camera (0.37%), USB (0.34%), Input (0.32%), Audio (0.26%), Network (0.23%), GPS (0.16%) And, Bluetooth (0.08%) is the lowest popular resource.

Additionally, we present the occurrences for system resources in relation to the categories of the Android apps at Table 4 the highest value is 17 for SQLite Database resource and category Multimedia followed by value 14 for USB and category Multimedia and then value 11 for Sensor, 8 for GPS and category Navigation. Moreover, the lowest values for Bluetooth resource across all versions categories. The Table shows that Multimedia category is the most frequent use of Database resources.

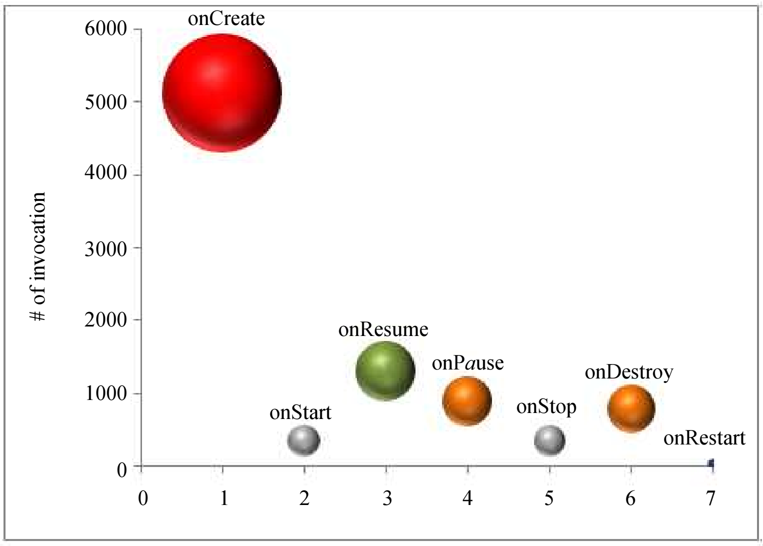

Fig. 4: Distribution of callback methods: Bubble chart showing the most popular callback methods over the data set

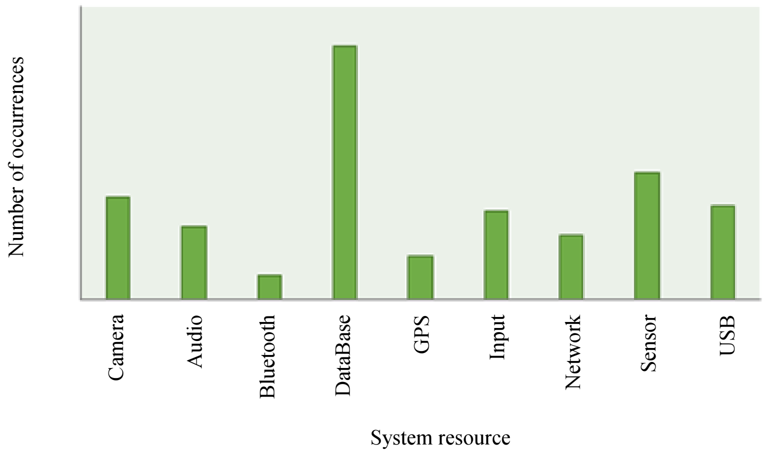

Fig. 5: Distribution of System Resources: showing the most popular System Resources over the data set

Table 3: Distribution of callback methods in terms of presence of activities

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()
Connectivity	(224) 4%	(28) 1%	(59) 1%	(46) 1%	(21) 0%	(77) 1%	(4) 0%
Development	(167) 3%	(14) 0%	(38) 1%	(22) 0%	(9) 0%	(18) 0%	(0) 0%
Games	(392) 7%	(22) 0%	(107) 2%	(93) 2%	(21) 0%	(49) 1%	(1) 0%
Graphics	(35) 1%	(0) 0%	(11) 0%	(3) 0%	(3) 0%	(7) 0%	(2) 0%
Internet	(888) 16%	(77) 1%	(261) 5%	(181) 3%	(80) 1%	(138) 2%	(4) 0%
Money	(227) 4%	(8) 0%	(79) 1%	(44) 1%	(10) 0%	(44) 1%	(0) 0%
Multimedia	(514) 9%	(49) 1%	(133) 2%	(108) 2%	(35) 1%	(100) 2%	(8) 0%
Navigation	(309) 6%	(26) 0%	(95) 2%	(68) 1%	(24) 0%	(35) 1%	(1) 0%
Phone and SMS	(150) 3%	(10) 0%	(46) 1%	(26) 0%	(6) 0%	(19) 0%	(0) 0%
Reading	(331) 6%	(16) 0%	(55) 1%	(48) 1%	(15) 0%	(46) 1%	(2) 0%
Science and Education	(550) 10%	(11) 0%	(115) 2%	(46) 1%	(22) 0%	(35) 1%	(0) 0%
Security	(196) 4%	(10) 0%	(50) 1%	(38) 1%	(11) 0%	(26) 0%	(2) 0%
Sports and Health	(215) 4%	(12) 0%	(28) 1%	(19) 0%	(13) 0%	(30) 1%	(0) 0%
System	(295) 5%	(26) 0%	(74) 1%	(42) 1%	(28) 1%	(59) 1%	(4) 0%
Theming	(70) 1%	(4) 0%	(14) 0%	(4) 0%	(3) 0%	(14) 0%	(0) 0%
Time	(317) 6%	(20) 0%	(68) 1%	(51) 1%	(32) 1%	(54) 1%	(2) 0%
Writing	(235) 4%	(13) 0%	(66) 1%	(49) 1%	(8) 0%	(29) 1%	(1) 0%

Table 4: Distribution of the system resources in terms of presence of activities

Category	Database	Sensor	Camera	USB	Input	Audio	Network	GPS	Bluetooth
Connectivity	(0) 0%	(1) 0.02%	(1) 0.02%	(1) 0.02%	(0) 0%	(2) 0.04%	(5) 0.09%	(0) 0%	(2) 0.04%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(4) 0.07%	(4) 0.07%	(0) 0%	(0) 0%	(0) 0%	(3) 0.05%	(0) 0%	(0) 0%	(1) 0.02%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(3) 0.05%	(1) 0.02%	(2) 0.04%	(1) 0.02%	(5) 0.09%	(1) 0.02%	(2) 0.04%	(0) 0%	(0) 0%
Money	(8) 0.14%	(0) 0%	(9) 0.16%	(1) 0.02%	(2) 0.04%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(17) 0.30%	(3) 0.05%	(4) 0.07%	(14) 0.25%	(2) 0.04%	(6) 0.10%	(0) 0%	(0) 0%	(0) 0%
Navigation	(1) 0.02%	(11) 0.20%	(0) 0%	(0) 0%	(1) 0.02%	(0) 0%	(3) 0.05%	(8) 0.14%	(0) 0%
Phone and SMS	(0) 0%%	(2) 0.04%	(2) 0.04%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(1) 0.02%	(2) 0.04%	(0) 0%	(0) 0%	(2) 0.04%	(0) 0%	(2) 0.04%	(0) 0%	(0) 0%
Science and Education	(8) 0.14%	(1) 0.02%	(1) 0.02%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 0.02%
Security	(4) 0.07%	(0) 0%	(0) 0%	(0) 0%	(3) 0.05%	(0) 0%	(1) 0.02%	(0) 0%	(0) 0%
Sports and Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(2) 0.04%	(0) 0%	(1) 0.02%	(1) 0.02%
System	(2) 0.04%	(0) 0%	(2) 0.04%	(2) 0.04%	(2) 0.04%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(1) 0.02%	(0) 0%	(0) 0%	(0) 0%	(1) 0.02%	(0) 0%	(0) 0%	(0) 0%
Writing	(4) 0.07%	(0) 0%	(0) 0%	(0) 0%	(1) 0.02%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5: Distribution of acquired API for each system resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
DataBase	(44) 85%	(0) 0%	(4) 8%	(4) 8%	(0) 0%	(8) 15%	(0) 0%	(35) 67%
Sensor	(26) 100%	(0) 0%	(2) 8%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 4%
Camera	(4) 19%	(1) 5%	(7) 33%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(17) 81%
USB	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Input	(6) 33%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Audio	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Network	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
GPS	(1) 11%	(0) 0%	(1) 11%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 11%
Bluetooth	(2) 40%	(1) 20%	(1) 20%	(0) 0%	(0) 0%	(1) 20%	(0) 0%	(2) 40%

Table 6: Distribution of released API for each system resource

Resource	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Database	(0) 0%	(0) 0%	(0) 0%	(2) 4%	(0) 0%	(8) 15%	(0) 0%	(0) 0%
Sensor	(1) 4%	(0) 0%	(0) 0%	(1) 4%	(0) 0%	(0) 0%	(0) 0%	(1) 4%
Camera	(1) 5%	(0) 0%	(2) 10%	(12) 57%	(1) 5%	(3) 14%	(0) 0%	(12) 57%
USB	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Input	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Audio	(7) 47%	(0) 0%	(0) 0%	(3) 20%	(0) 0%	(1) 7%	(0) 0%	(2) 13%
Network	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
GPS	(1) 11%	(0) 0%	(0) 0%	(1) 11%	(0) 0%	(1) 11%	(0) 0%	(1) 11%
Bluetooth	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Moreover, The Resource Analyzer was run In order to find the most popular acquired/released resources. The results of this analysis is shown in Table 5 and 6. Table 5 shows the most popular callback methods that used by developers to acquire each system's resources. The results show that:

- *Database* was acquired mostly on the onCreate() callback method with the percentage of occurrence equal to 85% over 52 activities which used Database resource. Also, 8% used the onPause(), 8% the onResume() and 15% used the onDestroy(). Whereas, around 67% of activities acquired Database on OTHER methods
- *Sensor* was acquired on the onCreate() method with the percentage of occurrence equal to 100% over 26 activities which used Sensor resource. Also, 8% used by the onResume(). Whereas, around 4% of activities acquired Sensor on OTHER methods
- *Camera* was acquired mostly on OTHER methods with the percentage of occurrence equal to 81% over 29 activities which used Camera resource. Also, 33% used the onResume(), 19% used the onCreate() and 5% used the onStart()
- *Input* was acquired mostly on the onCreate() with the percentage of occurrence equal to 33% over 18 activities which used Input resource
- *GPS* used 11%s of activities over 9 activities which acquired GPS resource in the onCreate(), onResume() and OTHER methods
- *Bluetooth* was acquired mostly on the onCreate() and OTHER methods callback methods with percentage of occurrence equal to 40% over 5 activities which used Bluetooth resource. Also, 20%s used the onStart(), onResume() and onDestroy()
- *USB*, Audio and Network resources had nothing of the percentages of acquired

However, Table 6 shows the results of the most popular callback method used to release resources. The results show that:

- *Database* was released mostly on the onDestroy() callback method with the percentage of occurrence equal to 15% over 52 activities which used Database resource. Also, 4% used the onPause()
- *Sensor* used 4%s of activities over 26 activities which released the onCreate(), onPause() and OTHER methods
- *Camera* was released mostly on the onPause() method with the percentage of occurrence equal to 57% over 29 activities which used Camera resource. Also, 5% used the onCreate(), 10% used the onResume(), 5% used the onStart(), 14% used the onDestroy(). whereas, around 57% of activities released Camera on OTHER methods
- *Audio* was released mostly on the onCreate() with the percentage of occurrence equal to 47% over 18 activities which used Audio resource. Also, 20% used the onPause() and 7% used the onDestroy(). Whereas, around 13% of activities released Audio on OTHER methods
- *GPS* used 11%s of activities over 9 activities which released GPS resource in the onCreate(), onPause(), onDestroy() and OTHER methods
- *USB*, *Input*, *Network* and *Bluetooth* resources had nothing of the percentages of released

In order to obtain more supportive results, the correctly acquired and released percentages was decided depending on Android documentation information. For each system's resources, the callback methods which is responsible to acquired and released resource were decided. Then, the value of percentages for these callback methods were decided as the correctly percentages of acquired and released the resource. On the other hand, the average of wrongly acquired and released percentages was computed by finding the averages of the callback methods percentages that were registered to acquire and to release but did not have a responsibility to do that. Figure 6 and 7 show these comparisons of correctly/wrongly acquired and released percentages for the system's resources. These results show that:

- *Database* resource should be acquired at onCreate() and released at onPause() methods (Activity android developers, n.d.). Our result shows that about 85% activities used onCreate() to acquire Database resource and 4% of activities used onPause() to release Database resource correctly. However, the average of wrongly acquired is equal to 25%. It includes 67% of activities used OTHER method, 15% used onDestroy() and 8% used onResume() or onPause() to acquire Database resource. Additionally, the average of wrongly released is equal to 15% of activities used onDestroy() to release Database resource
- *Sensor* resource should be acquired at onResume() and released at onPause() (Activity android developers, n.d.). Our result shows that about 4% of activities used onResume() to acquire Sensor resource and 4% of activities used onPause() to release Sensor resource correctly. However, the average of wrongly acquired is equal to 52%. It includes 100% of activities used onCreate() method and 4% used OTHER to acquire Sensor resource. Additionally, the average of wrongly released is equal to 4%. It includes 4% of activities used onCreate() method and also 4% used OTHER to release Sensor resource
- *Camera* resource should be acquired at onResume() and released at onPause() (Activity android developers, n.d.). Our result shows that about 33% of activities used onResume() to acquire Camera resource and 57% of activities used onPause () to release Camera resource correctly. However, the average of wrongly acquired is equal to 35%. It includes 81% of activities used OTHER method, 19% used onCreate() and 5% used onStart() to acquire Camera resource. Additionally, the average of wrongly released is equal to 13%. It includes 14% of activities used onDestroy() and 57% used OTHER method to release Camera resource
- *USB* should be acquired at onResume() and released at onPause() (Activity android developers, n.d.). Our result showed that there are no occurrences of acquired or released USB resource in our data set
- *Input* resource should be acquired at onCreate() and released at onPause() (Activity android developers, n.d.). Our result showed that about 33% of activities used onResume() to acquire Input resource. However, there are no occurrences of released Input resource in our dataset
- *Audio* resource should be acquired at onCreate() and released at onPause() (Activity android developers, n.d.). Our result showed that there are no occurrences of acquired Audio resource at onCreate() in our dataset and 20% of activities used onPause() to release Audio resource correctly. However, the average of wrongly released is equal to 30%. It includes 47% of activities used onCreate() method, 7% used onDestroy() and 13% used OTHER to release Audio resource
- *Network* resource should be acquired at onCreate() and released at onPause() (Activity android developers, n.d.). Our result showed that there are no occurrences of the acquired or released Network resource in our dataset
- *GPS* resource should be acquired at onCreate() and released at onPause() (Activity android developers, n.d.). Our result showed that about 11% of activities used onCreate() to acquired and also11% of activities used onPause() released GPS resource

correctly. However, the average of wrongly acquired is equal to 11%. It includes 11% of activities used onResume() method and 11% used OTHER to acquire GPS resource. Additionally, the average of wrongly released is equal to 11%. It includes 11% of activities used onCreate()method and 11% used OTHER to release GPS resource.

- *Bluetooth* resource should be acquired at onCreate() and released at onPause() (Activity android developers, n.d.). Our result showed that about 40% of activities used onCreate() to acquire Bluetooth resource correctly and there are no occurrences of released Bluetooth resource at onPause() method. However, the average of wrong acquired is equal to 25%. It includes 40% of activities 40% used OTHER method, 20% used onPause() and onResume() to acquire Bluetooth resource

Part III: The Nature of the Code Implemented Inside Callback Methods

The third part of our analysis focuses on the nature of code inside the most important callback methods onPause(), onStop() and onDestroy(). We divided the nature of code into three categories. The first group is that code used for releasing actions; the second is associated with the code used for database actions; and the third code is related to managing threading actions. We also considered the second and third categories (database and threading actions) as a long running and heavy code actions doc.

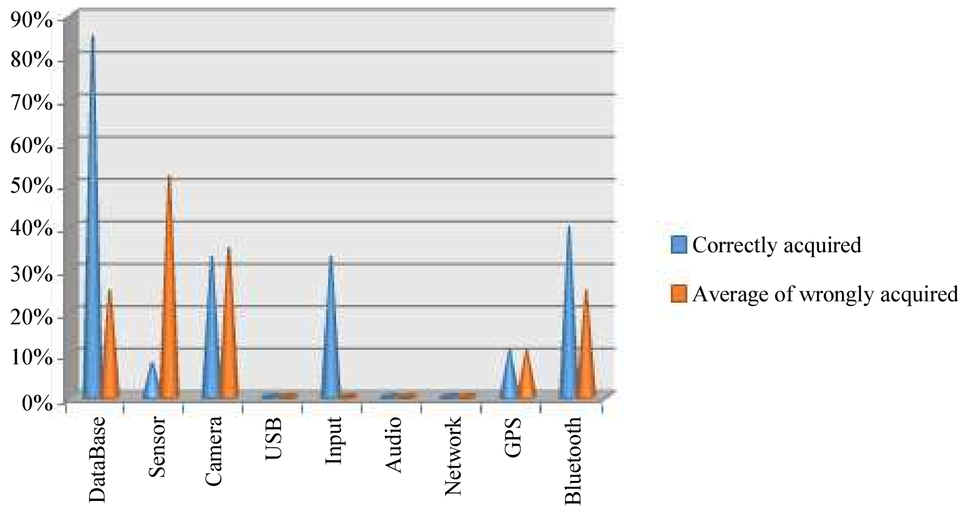

Fig. 6: The average of correctly and wrongly acquired of system's resources

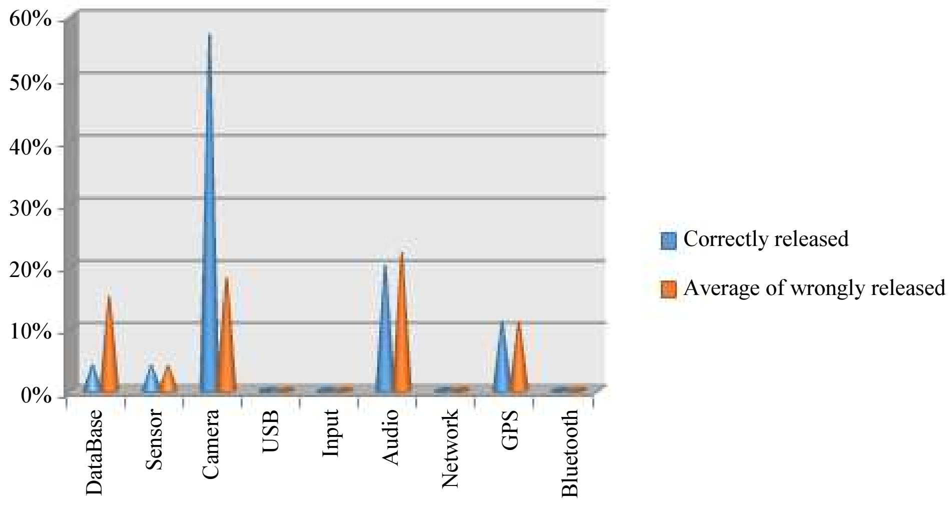

Fig. 7: The average of correctly and wrongly released of system's resources

Table 7: Long Running code analysis on OnPause(), OnStop() and OnDestrory() methods

#	Category Name	onPause()	onStop()	onDestroy()
I	Releasing Resources actions	-	(50) 15%	(214) 27%
II	Database actions	(104) 12%	(35) 10%	(41) 5%
III	Threading actions	(43) 5%	(8) 2%	(45) 6%
	Total	(147) 17%	(341) 27%	(780) 38%

Using SAALC, we applied analysis using keywords representing each of these code categories for on onPause(), onStop() and onDestroy() methods. However, due to the onPause() callback method used to release system resources, we analyzed the data set for onPause() callback method over the second and third categories (long running code) only to avoid the conflict with the main purpose of its usage which is releasing actions.

The results of this analysis are shown in Table 7. It shows that the total percentages of all three categories inside onPause() is (147)17%. This includes the percentages of long running code (104) 12% for database actions; and (43) 5% for threading actions. On the other hand, the total percentage for the three categories at onStop() is (341)27%. This includes (50)15% for releasing actions; (35) 10% for database actions; and (8)2% for threading actions. Further, the total of percentages for the nature of code inside onDestroy() is (780) 83%. this includes (214) 27% for releasing actions, (41)5% and (45) 6% for long running code (database and threading actions respectively). We discuss these results in more detail in Section 4.

Discussion

In this section we provide a discussion of the experimental results in terms of our research questions.

RQ1: To what extent Android developers utilize the lifecycle callback methods in developing mobile apps?

Answering RQ1 gives us the first indication about Android lifecycle callback methods' utilization in actual real-world apps. The results in Result section show that the onCreate() callback method is the one that is mostly implemented (92%) in the activities of the selected apps. This is not surprising since the onCreate() method is the main method to start and setup Android activities (Activity android developers, n.d.). Implementing the onCreate() callback method is important to initialize the user interface of the activity as well as its various data binding operations. Moreover, app developers use the onCreate() method to do all normal operations of creating views and activity setup.

The onResume() callback method is implemented by 23% of the total number of app activities extracted. It is usually called when the activity is in the foreground and about to start interacting with users' interactions (Activity android developers, n.d.). Additionally, it is used for acquiring system resources among other services by developers. Thus, we can assume that such percentage of utilizing the onResume() method is reasonable.

The onPause() callback method usage percentage is implemented by 16% of activities and is normally used when an activity is about to go to the background (Activity android developers, n.d.). More specifically, it is used to commit unsaved changes, release system resources, stop animations and other processes that can consume the CPU. We can also conclude here that such usage percentage is also reasonable. This is because not all actives need to deal with releasing system resources or to persist data.

On the other hand, the onRestart() callback method is rarely implemented (1%) in the activities. The onRestart() callback method is normally used after an activity has stopped and before it is started back again (Activity android developers, n.d.). Additionally, it is used to acquire a row cursor objects if a developer has already deactivated it at onStop() method (Activity android developers, n.d.). Cursor objects provide random read-write access to the result set returned by a database query (Activity android developers, n.d.). Accordingly, we can assume that the usage percentage of onRestart() method (1%) can also be reasonable since it has limited usage scenarios.

The onStart() and onStop() callback methods both have usage percentage implementation in activities of (6%. The onStart() callback method is normally called when an activity is newly created and becoming visible to the user (Activity android developers, n.d.). However Google documentation do not provide clear description of when to use this method.Whereas, The onStop() callback method is normally used to save app data to permanent storage and to execute long running code (Activity android developers, n.d.). According to (Activity android developers, n.d.)), developers must execute long running code at onStop() instead of onPause() method. This is due to the fact that onPause() method must be executed quickly so that other activities can start seamlessly. Thus 6% implementation for onStart() and onStop() are reasonable percentages as well.

The onDestroy() callback method has implementation percentage of 14%. The onDestroy() callback methods are normally called before an activity is destroyed (Activity android developers, n.d.). It acts as the last chance for developers to free resources and threads that are associated with the activity before it is removed from memory (Activity android developers, n.d.). However, according to the study by (Franke *et al*., 2012), the onStop() and onDestroy() methods may not be called by the system in cases were the system is very low in resources (battery and memory). In such a situation, developers may face a dilemma. This is because both the onStop() and onDestroy() methods are normally used to execute long-running code such as data persistence. We will have more on this in the discussion for RQ3.

RQ2: Did each Android developer correctly acquire and release Android system resources?

The averages of correctly acquiring and releasing of the nine system resources mentioned at Result section are 23% and 11% respectively. However, the averages of wrongly acquiring and releasing system resources are 16% and 8% respectively. Such results show that a relatively large percentage of system resources are not correctly released by the developers in the activities for the apps that we selected for analysis. According to the study by (Franke *et al*., 2012), this will lead to incorrect behavior of Android apps as well as memory leaks and run-time errors. Furthermore, we argue that due to this high number of misuses, the developers seem to be not fully aware of the importance of correctly managing the system resources.

RQ3: What is the nature of code implemented inside onPause(), onStop and onDestroy() callback methods?

Regarding the analysis of the nature of code implemented inside onPause(),onStop and onDestroy() callback methods, we found that the onStop() method has a 12% of code that is considered to be long running code such as database and threading actions. Further the onDestroy() method has 11% of code is considered to be long running code. This is acceptable from the point of view of the Android official documentation (Activity android developers, n.d.). It is true that the study by (Franke *et al*., 2012) argue that the onStop() and onDestroy() methods may not be called, but this can only happen at very extreme cases when the Android system is very low in resources. And since the developers should not implement such long running code at onPause(), we can conclude that onStop() and onDestroy() are still the best place to implement such code. Regarding the nature of code inside onPause() callback method, we found this has 17% of code that we considered to be long running code. This is a problematic issue as discussed above since this will possibly block other activities from running seamlessly.

Our results show that Android developers, in general, appear to posses limited knowledge and awareness of the importance of writing an app that conforms to the lifecycle model. This will adversely affect their apps' reliability and performance. Further, we argue that Android documentation needs to be more useful, complete and clear in describing how developers should apply activity callback method and system resources. We also argue that Android developers and more specifically the novice, should get more help in terms of managing system resources from the development environment, namely the Android Studio. Having the Android Studio automatically manage lifecycle conformance wherever feasible can be beneficial, especially for less expert developers.

Android developers can use our findings to gain deep insights about their Android apps development and pay particular attention to their use of callback methods and resource management. Moreover, software researchers can use our findings to provide better support to developers by providing analysis tools, in order to help developers of building more robust apps. In the future, we aim to expand our study by analyzing more apps from different platforms such as iOS and by adding further mining and analysis techniques to our approach.

Threat to Validity

Threats to internal validity include that we automatically identified apps which contain system resources using imported API packages in the .java files. Sometimes developers insert unused resources that are not called. Also, we used the fields' names according to the type of API to check where the resource acquired and released inside the methods. Further, the limitation in our tool is that it is currently unable to recognize the inheritance mechanism and code hierarchy that may exist in the app code. Threats to external validity are present as the results may not generalize to other kinds of applications, the applications selected may not be representative and other app store repositories may provide apps with different characteristics. It is important to note that in our data set, some categories, such as Graphics, were small and thus cannot be representative sample. However, we have tried to mitigate this by selecting a range of apps and investigating 5577 Android activities from842 out of 2001 apps from F-Droid, which is one of the largest repositories of open-source Android apps.

Conclusion

The Android mobile app activity lifecycle model is very important to understand in order to develop robust apps. With an ever-growing app community, activity lifecycle holds more importance to ensure that apps are adequately reliable and robust. Our study is the first study to explore the activity's lifecycle callback methods usage in the Android development community. We built a tool called SAALC to analyze 5577 activities residing in 842 Android apps from F-Droid repository. We analyzed activities to collect statistics about the utilization of each callback method; the averages of correct and wrong acquired/released system resources; and the nature of long running process inside onPause(), onStop() and onDestroy() callback methods. Our findings can be summarized as follows:

- The percentages of occurrences of callback methods are about 1% of the activities used onRestart() method, 6% used onStop(), 23%usedonResume(), 16% used onPause() and 14% used onDestroy(). The most occurrences for onCreate() callback methods for about 92% of activities
- Only about 3% of activities contain Camera, Audio, Bluetooth, Database, GPS, Input, Network, Sensor and USB system's resources
- The average % of callback method code that wrongly acquires a system's resource is 16%, whereas the average of wrongly released is 8%. This will adversely influence the app reliability
- About 17% of activities used long running code inside onPause() callback methods and this will adversely influence the app performance
- 27% of activities used releasing and long running code inside onStop() callback method, whereas 38% used inside onDestroy()

We propose that developers need further guidance about correct use of lifecyle-related callback method usage and code. We also propose that improved development tool support for Android developers is needed to provide improved automatic detection of lifecycle resource management problems in apps.

Author's Contributions

Noura Hoshieah: Contributed in theoretical aspect of this study, downloaded the apps source code, developed the tool, performed data collection and data analysis.

Dr. Samer Zain: Advised research process, research design, data collection and analysis, and manuscript writing.

Dr. Norsaremah Salleh: Advised in research design and manuscript writing.

Dr. John Grundy: Conceived of the presented idea.

Ethics

This research is original and not published elsewhere. The authors confirm that they have read and approved the manuscript and there is no conflict of interest. Further, the authors confirm that there are no ethical issues involved.

References

Activity android developers, (n.d.). https://developer.android.com/reference/android/app/Activity.html

Bartel, A., J. Klein, M. Monperrus and Y.L. Traon, 2014. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. IEEE Trans. Software Eng., 40: 617-632.
DOI: 10.1109/TSE.2014.2322867

Batyuk, L., M. Herpich, S.A. Camtepe, K. Raddatz and A.D. Schmidt *et al*., 2011. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. Proceedings of the 6th International Conference on Malicious and Unwanted Software, Oct. 18-19, IEEE Xplore Press, Fajardo, Puerto Rico, pp: 66-72.
DOI: 10.1109/MALWARE.2011.6112328

Danphitsanuphan, P. and T. Suwantada, 2012. Code smell detecting tool and code smell-structure bug relationship. Proceedings of the Spring Congress on Engineering and Technology, May 27-30, IEEE Xplore Press, Xian, China, pp: 1-5.
DOI: 10.1109/SCET.2012.6342082

Dehlinger, J. and J. Dixon, 2011. Mobile application software engineering: Challenges and research directions. Mobile Software Eng., 2: 29-32.

Desnos, A., 2012. Android: Static analysis using similarity distance. Proceedings of the 45th Hawaii International Conference on System Science, Jan. 4-7, IEEE Xplore Press, Maui, HI, USA, pp: 5394-5403.
DOI: 10.1109/HICSS.2012.114

F-Droid. Free and Open Source Android App Repository, (n.d.). https://f-droid.org/

Feng, Y., S. Anand, I. Dillig and A. Aiken, 2014. Apposcopy: Semantics-based detection of android malware through static analysis. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Nov. 16-21, ACM, Hong Kong, China, pp: 576-587.
DOI: 10.1145/2635868.2635869

Franke, D., C. Elsemann, S. Kowalewski and C. Weise, 2011. Reverse engineering of mobile application lifecycles. Proceedings of the 18th Working Conference on Reverse Engineering, Oct. 17-20, IEEE Xplore Press, Limerick, Ireland, pp: 283-292.
DOI: 10.1109/WCRE.2011.42

Franke, D., S. Kowalewski, C. Weise and N. Prakobkosol, 2012. Testing conformance of life cycle dependent properties of mobile applications. Proceedings of the IEEE 5th International Conference on Software Testing, Verification and Validation, Apr. 17-21, IEEE Xplore Press, Montreal, QC, Canada, pp: 241-250.
DOI: 10.1109/ICST.2012.104

Haotian, Z. and L. Shu, 2013. Java source code static check eclipse plug-in based on common design pattern. Proceedings of the 4th World Congress on Software Engineering, Dec. 3-4, IEEE Xplore Press, Hong Kong, China, pp: 165-170. DOI: 10.1109/WCSE.2013.30

Joorabchi, M.E., A. Mesbah and P. Kruchten, 2013. Real challenges in mobile app development. Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Oct. 10-11, IEEE Xplore Press, Baltimore, MD, USA, pp: 15-24. DOI: 10.1109/ESEM.2013.9

Khatoon, S., A. Mahmood and G. Li, 2011. An evaluation of source code mining techniques. Proceedings of the 8th International Conference on Fuzzy Systems and Knowledge Discovery, Jul. 26-28, IEEE Xplore Press, Shanghai, China, pp: 1929-1933. DOI: 10.1109/FSKD.2011.6019877

Lamba, Y., M. Khattar and A. Sureka, 2015. Pravaaha: Mining android applications for discovering API call usage patterns and trends. Proceedings of the 8th India Software Engineering Conference, Feb. 18-20, ACM, Bangalore, India, pp: 10-19. DOI: 10.1145/2723742.2723743

Number of apps, (n.d). Number of apps in leading app stores. https: //www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores

Panichella, S., V. Arnaoudova, M. Di Penta and G. Antoniol, 2015. Would static analysis tools help developers with code reviews? Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering, Mar. 2-6, IEEE Xplore Press, Montreal, QC, Canada, pp: 161-170. DOI: 10.1109/SANER.2015.7081826

Payet, E. and F. Spoto, 2012. Static analysis of android programs. Inform. Software Technol., 54: 1192-1201. DOI: 10.1016/j.infsof.2012.05.003

Ramos, A., 2016. Evaluating the ability of static code analysis tools to detect injection vulnerabilities. http://www8.cs.umu.se/education/examina/Rapporter/AlexanderRamos.pdf

Schmidt, A.D., R. Bye, H.G. Schmidt, J. Clausen and O. Kiraz *et al.*, 2009. Static analysis of executables for collaborative malware detection on android. Proceedings of the IEEE International Conference on Communications, Jun. 14-18, IEEE Xplore Press, Dresden, Germany, pp: 1-5. DOI: 10.1109/ICC.2009.5199486

Wasserman, A.I., 2010. Software engineering issues for mobile application development. Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, Nov. 07-08, ACM, Santa Fe, New Mexico, USA, pp: 397-400. DOI: 10.1145/1882362.1882443

Zein, S., N. Salleh and J. Grundy, 2016. A systematic mapping study of mobile application testing techniques. J. Syst. Software, 117: 334-356. DOI: 10.1016/j.jss.2016.03.065

Zein, S., N. Salleh and J. Grundy, 2017. Static analysis of android apps for lifecycle conformance. Proceedings of the 8th International Conference on Information Technology, May 17-18, IEEE Xplore Press, Amman, Jordan, pp: 102-109. DOI: 10.1109/ICITECH.2017.8079982

Zhongyang, Y., Z. Xin, B. Mao and L. Xie, 2013. DroidAlarm: An all-sided static analysis tool for android privilege-escalation malware. Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, May 08-10, ACM, Hangzhou, China, pp: 353-358. DOI: 10.1145/2484313.2484359