Android Application Runtime Memory Performance Analysis Framework

By: Abebaw Degu

A Thesis Submitted to

Department of Computing

School of Electrical Engineering and Computing

Presented in Partial Fulfillment for the Degree of Master of Science in Computer Science and Engineering

Office of Graduate Studies

Adama Science and Technology University

June, 2018

Adama, Ethiopia

Android Application Runtime Memory Performance Analysis Framework

By: Abebaw Degu

Name of Advisor: Mesfin Abebe (PhD.)

A Thesis Submitted to

Department of Computing

School of Electrical Engineering and Computing

Presented in Partial Fulfillment for the Degree of Master of Science in Computer Science and Engineering

Office of Graduate Studies

Adama Science and Technology University

June, 2018

Adama, Ethiopia

DECLARATION

I hereby declare that this MSc. thesis is my original work and has not been presented as a partial degree requirement for a degree in any other university, and that all sources of materials used for the thesis have been dully acknowledged.

Name: Abebaw Degu

Signature:_____

This thesis has been submitted for examinaiton with my approval as thesis advisor.

Name: Mesfin Abebe (PhD.)

Signature:_____

Date of Submition:_____

Approval of Board of Examiners

We, the undersigned, members of the Board of Examiners of the final open defense by _____have read and evaluated his/her thesis entitled "_____" and examined the candidate. This is, therefore, to certify that the thesis has been accepted in partial fulfillment of the requirement of the Degree of ……

_____	_____	_____
Supervisor/Advisor	Signature	Date
_____	_____	_____
Chairperson	Signature	Date
_____	_____	_____
Internal Examiner	Signature	Date
_____	_____	_____
External Examiner	Signature	Date

Acknowledgment

Firstly, I would like to express my sincere gratitude to my advisor Dr. Mesfin Abebe for the continuous support of my MSc Study, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of the study, research and writing of this thesis.

Besides my advisor, I would like to thank my classmates and lab mates for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had for the last two years.

I thank God for protecting me in life and guides me in the path of wisdom. His patience and protection enable me to be where I am now.

The last but not the least, I would like to thank my wife, for her unwavering support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without her.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ADB	Android Debug Bridge
AOT	Ahead of Time
API	Application Program Interface
APK	Android Package Kit
App	Application
ART	Android Runtime
DEX	Dalvik Executable
DSL	Domain Specific Language
DVM	Dalvik Virtual Machine
FCG	Function Call Graph
GC	Garbage Collection
GPS	Global Positioning System
GPU	Graphical Processing Unit
GUI	Graphical User Interface
IDE	Integrated Development Environment
JIT	Just In Time
JVM	Java Virtual Machine
LMK	Low Memory Killer
OOM	Out Of Memory
OS	Operating System
SDK	System Development Kit

Abstract

Application developers are facing various challenges during the development of applications for smartphone mobile devices. Smartphone devices have limited memory resources that restrict developers from developing computation-intensive applications. To fulfill the needs of end users, developers must take these memory limitations into account; they should try for their applications memory performance optimization.

This study provides a source code level optimization of memory utilization and automated means of testing the memory for the Android application development. Sub-areas are empirically examined in detail: the impacts of resource leak patterns (AsyncTask, Thread, Handler, Static Variable, Static Variable to an Instance of Inner Class, Singleton, and Static View Activity) on memory, and the effects of these patterns on the total memory usage of an app. An empirical evaluation was conducted to see the effects of these patterns on emulators and real-time mobile devices with the two core runtime libraries (Dalvik and ART).

The result shows that memory resource leakage can be avoided using the best practices suggested in the Android framework. Leak pattern can impose huge memory consumption when it runs with other activities and processes in the application. The leakage through long-running activity on the running application measures the average of 6.02 MB and 4.71 MB which is larger than the leak resulting from static reference (4.37 MB and 2.51 MB) in the emulator and real devices respectively. Moreover, the leakage due to long-running activity leak pattern and static view activity leak pattern lasts throughout the lifetime of the application.

Keywords: Android Application, Memory Leak Pattern, Performance Analysis, Memory Performance, Source code, Test Automation, Runtime

CHAPTER ONE

INTRODUCTION

Android is a Linux based operating system which is primarily designed for touchscreen mobile devices such as smartphone devices, and tablets that are widely available on the market. Android is a powerful operating system that provides comfortable applications for users, and it is opensource which means it is free so that anyone can use and modify it.

Nowadays smartphones and tablet users and market needs are explosively increasing. Developers are trying to develop an application that satisfies customer's needs. Android platform app is currently having the largest market share and is expected to continue its explosive growth in the future [1]. But, most of the application is not usable to every phone as it consumes more memory and becoming too sluggish.

Our daily lives almost have become dependent on mobile devices such as smartphones and tablets. These devices enable us to use apps that provide us helpful and service intensive computation which makes our lives easy. Unfortunately, these devices, unlike computers are limited in terms of their memory capacity. Despite its limited resources, some programming practices contribute its part for the inefficient use of our device's memory.

To maximize the usability of the apps that are going to be developed, developers do not have sufficient guidance to the most useful practices and to improve the resource utilization of their implementations [2]. Even though, there is the advancement in hardware technologies poorly designed apps are not stopped from needlessly consuming the resources.

Mobile apps have been in the 1990s as they were created by Nokia and Motorola [3]. These apps have led to the development of various apps in different categories and hosted in different App stores. The apps from the most popular App stores, Google Play and Apple store, have been downloaded more than 125 billion times, but performance issues are among the bottlenecks that affect app's rating [4]. The developers of these apps are currently facing the challenges of memory as they receive complaints from users.

1.1 Motivation

Based on the statistics conducted by Dogtieve [5], the total number of app downloaded in 2016 exceeds that of 2015 by 40 billion, which is 90 billion. As of June 2017, the total number of Android apps on Google play is 3 million. It implies that the current rate of growth is more than 1,300 apps per day. Many smartphone and tablet companies sell devices that run Android OS, hence android apps developed are much larger than Apple's iOS.

The reason for choosing mobile application also comes from the statistics [6] which states customers prefer apps (85%) ahead of mobile sites because apps are seen to be more convenient (55%), faster (48%) and easier to browse (40%). This survey indicates that customers expect mobile apps, with 42% to load faster than a mobile website; 36% expect to load at the same speed; whereas 23% expect mobile web to load faster than an app. On average smartphone owners expect apps to load in two seconds while 28% said that apps should load in one second or less. This indicates that users have less patience with apps than they do with mobile webs. In addition to this, mobile users expect mobile sites to load in four seconds. The most common problems that have been identified are mobile app crashing or freezing 62%, being slow to launch 47%, unable to launch at all 40%.

Among customers, 79% would retry an app once or twice if the first attempt fails whereas 16% would try more than two attempts. In iOS, the most common user complaints were lack of features 26%, frequent crashing 23%, and poor system design 22%. In the same manner, android users suffer from similar problems with crashing being the main complaint 33%, the app not working as intended 26%, and lack of features 25%. Beyond this android leads the market share for smartphone mobiles and tablet OS with 69.18% followed by iOS at 25.02% as of September 2016.

According to the survey conducted by Sherman [7], 53% of users uninstalled a mobile application with severe issues of crashes, freezes or errors. Similarly, 36% users will stop using apps due to higher battery consumption [7] [8] which likely causes device overheating. On the other hand, Fakhruddin [8] stated that almost 30% of Android apps are deleted within 10 minutes of download due to the requirements of large memory size or taking too much storage space.

1.2 Statement of the Problem

While using android applications in smartphones, the following problems are observed based by the researcher on the current usage statistics of these apps:

The problems identified are the apps frequently crashes, freezes, and slow to launch. In addition to this, there is abnormal memory utilization. Some applications suffer from high memory allocation which leads to the sluggishness and Out of Memory (OOM) error after the long-lasting execution of running apps. Moreover, there is a dificulty in representing test cases for that hinders execution of the testing of the application memory consumptions automatically by representing of the leak patterns of android applications.

On the other hand, developers do not have sufficient guidance to follow for developing apps that are efficient in memory utilization of applications prior to releasing or publishing it to the customers and App stores.

The focus of the Android application developer is to provide an app that is highly needed and will be rated by a customer. The users of these apps may download the apps and see whether it is convenient to use based on their memory and other performance capacity. If those apps are unable to use memory efficiently and unable to load faster, the users will simply uninstall and look for other apps, as a result reducing the developer's App rating.

The study focuses on assessing the performance impacts of resource leak patterns on memory utilization, automation of memory testing, and producing the guideline to provide an insight for the developers to improve their programming style. To answer the problems stated above and meet the objective of this research the following research questions are answered:

RQ1: What are the issues that lead to inefficient memory utilization?

RQ2: How the test of memory utilization can be automated?

RQ3: How the leak patterns affect the memory performance of the running application?

RQ4: How the memory utilization of Android applications can be improved by automated test of the memory?

1.3 Significance of the Study

The result of this study will benefit application developers in the following major aspects: -

- Developers will be aware of memory consuming programming practices and will remove OOM problems.
- Developers will get higher app rating due to the performance improvement.
- The result of this study will serve as an input for Android application developers.

1.4 Objective of the Study

1.4.1 General Objective

The general objective of this study is to develop a framework for evaluating memory consumption of resource leak performance patterns using automated memory test.

1.4.2 Specific Objective

To realize abovementioned general objective the following specific objectives are identified:

- Conduct a literature review on android applications memory consuming practices and analysis techniques.
- Identify different issues that affect the memory performance of the application.
- Identify analysis techniques that are appropriate for this analysis task.
- Develop benchmarking app that visualizes resource leak patterns which affect the performance of applications.
- Start memory performance testing on the developed apps.
- Analyze memory performance of the developed apps.
- Optimize source codes that lead to high resource consumptions.
- Restart memory performance testing on the optimized source code and compare the result with the original one.

1.5 Scope and Limitation

Smartphones have limited amounts of resources when compared to computers. Moreover, most of the mobile applications memory utilization is not efficient. This study only focuses on memory

resource leak patterns impact on memory utilization and the automation of the memory usage testing of the apps. To broaden the addressability of the problems, real-time apps from App stores should be evaluated for their memory consumption. However, this study's evaluation doesn't consider those apps from App stores. Because the apps downloaded from App Stores requires de-compilation to get the source code and to trace the possible leakages in the source code. However, the decompiled APK files don't have a Gradle file that holds the dependencies. On the other hand, to test the performance of the apps, Gradle is required for apps to be executed. Moreover, studying and understanding of the de-compiled APKs source codes to represent the activities with test cases requires several months or years depending on the number of available classes within the downloaded APK.

1.6 Research Methodology

1.6.1 Literature Review

A literature review has been conducted to assess concepts, tools, techniques, and applications of performance analysis and to get a good understanding of the domain knowledge and the problem area. Relevant documents and android application related websites were reviewed to get a thorough understanding of how memory is related to performance within smartphones and how programming practices pave the way for efficient utilization of these resources.

1.6.2 Benchmarking

Using previously developed apps or developing the new applications with the leak behavior which helps for the evaluation of the memory leak patterns impact on memory consumption is called benchmarking. The apps which are used or developed helps to assess the possible improvements of the memory consumption after conducting source code optimization.

1.6.3 Tools

Monkey Runner: In this study, this tool is used to write python program using Jython that controls the emulator and the device outside of the Android code. Moreover, it is used to test the applications and devices at the functional/framework level and for starting the performance analysis script.

Python 2.7: Since it provides the functionality of debugging and code optimization, in this study, it is used to write the script for the automation test.

Jython: Is a language used by Monkey runner so that the script is written using the Jython syntax.

Espresso: It used to write test cases to simulate user interaction.

Groovy: Is used to write a custom plugin to add a custom task such as automation test to a project.

Android Studio: Is used to write, execute, and test the benchmarking apps.

Finally, Microsoft Office Word application was used for document preparation and *LATEX Beamer* presentation was used to present the result of the study.

1.6.4 Code Optimization

It is a technique which involves code rearrangement or restructuring that helps in enhancing the performance of Android applications.

1.7 Organization of the Thesis

The thesis is organized into seven Chapters, including this Introduction Chapter. Chapter Two presents the literature review and related work of the study. The literature review briefly explains about memory management and analysis techniques, resource leaks, Android and Android applications, performance optimization on Android applications, and applications of performance optimization. Chapter Three presents the Methodology of the study. The methodology of the study shows the overall procedures of the study in detail. Chapter Four discusses the high-level architecture of the proposed solution. In this chapter, the detailed components, flows, and interaction between components are described. Chapter Five presents the implementation of the proposed solution. In this section, resource leak patterns and optimized version of these leaks, and automation test mechanisms are implemented and described in detail. Chapter six presents the evaluation, result, and discussion of the study. This part describes in detail the memory consumption with the presence and absence of the leak patterns by using emulators and actual device. Moreover, the detailed analysis is performed based on the experiments results that are found. The last Chapter presents the conclusions, and future work of the study. Furthermore, it shows how the research questions are answered in the study.

CHAPTER TWO

LITERATURE REVIEW AND RELATED WORK

Several published papers; thesis works, and websites were reviewed on different topics that are related to this study such as overview of Android, Android architecture, Android applications, Android memory management, garbage collection, allocation and reclamation of application memory, memory resource leakage, object reference, memory leak pattern, performance analysis and optimization of android applications, memory utilization, energy consumption, network traffic and performance, GUI rendering, and related works of the study.

2.1 Overview of Android

Android is a comprehensive, opensource platform, and Linux-based operating system primarily designed for devices such as smartphones and tablet computers which is released in September 2008 by Open Handset Alliance, led by Google, and other companies. It is an open source so that anyone can get the access and modify it. In Android, there are multimillion of apps on the market that can make our life easy and comfortable as it provides a computation intensive application that would be unable or take too much time to be executed by humans.

Android provides a platform for application development, hence, after developers develop applications for Android, their applications could run on different devices supporting Android operating system. The reason that makes android operating system preferable over others is that it is opensource, have larger developer community, increased marketing, provide inter-app integration, reduce the cost of development, higher success ratio, and rich development environment. The one thing that makes it different from others is that it doesn't control its own lifecycle [9] and has plenty of callback function instead of the main function.

2.1.1 Android Architecture

Android architecture has five layers: Linux kernel, android runtime, libraries, application frameworks, and applications as depicted in Figure 1. In the architecture ART, for Android version 5.0 (API level 21) or higher is used which can run many virtual machines on low memory devices [10]. The features of ART include: Ahead-of-time (AOT) and Just-in-time (JIT) compilation, enhanced garbage collection (GC), and improved debugging support. On the other hand, DVM uses Just In Time (JIT) compiler. This compiler is a software component which takes application

code, analyzes the code, and translates it into a form that runs faster. The application which runs on ART can run well on Dalvik, but the reverse doesn't work.

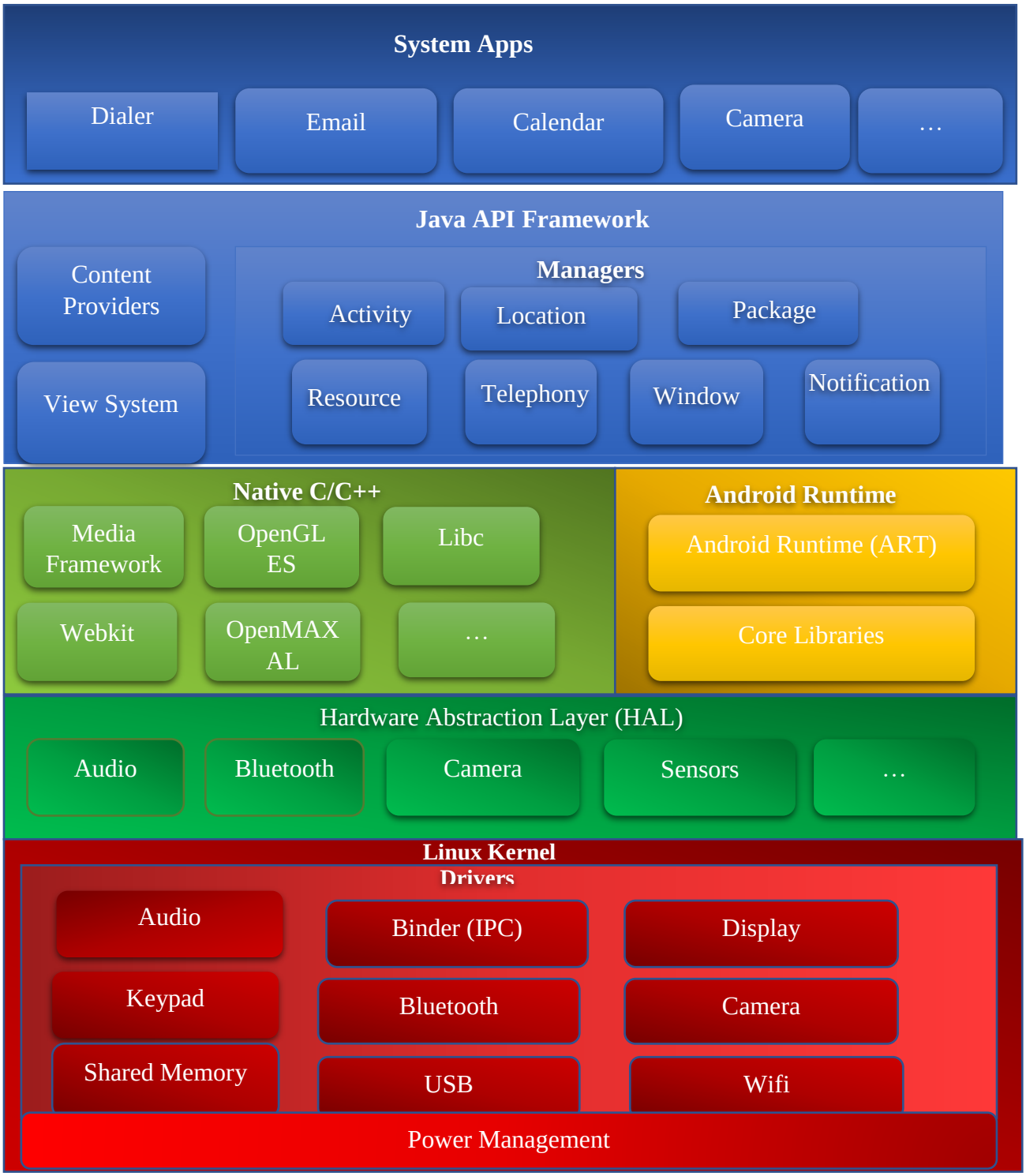


Figure 1: Android Architecture

The layers of the Android architecture which is depicted in Figure 1 is discussed as follows:

- ***Linux Kernel:*** it is a foundation of the Android platform. Moreover, it allows the Android to take advantage of critical security features and allows enables device manufacturers to develop drivers for well-known kernel.
- ***Hardware Abstraction Layer (HAL):*** it provides interfaces that can expose hardware device capabilities to the higher-level Java API framework.
- ***Android Runtime:*** the purpose of ART is to run multiple virtual machines on low memory devices through executing DEX files which is a byte code format primarily designed for Android which is optimized for minimal memory footprint.
- ***Native C/C++:*** services such as ART and HAL are built from native codes which require native libraries that is written in C and C++.
- ***JAVA API Framework:*** is used to expose native libraries functionality to applications. Moreover, it uses for making all the features of Android OS to make available to users through APIs.
- ***System Apps:*** it functions as apps for users and used to provide basic capability for developers to access from their own apps.

2.1.2 Android Applications

Android applications are usually developed using Java programming language with the help of Android software development kit (ADT). However, recently in Google IO 2017, Kotlin is introduced as the official language to develop Android applications.

An Android application consists of four components: Activities (provides in which a user can interact), Services (operate remote process and handling of long-running processes in the background), Broadcast receivers (to respond system-wide broadcast announcements), and Content providers (manages shared app data for queries) [11]. Acquiring and releasing of resources are performed at different stages of the life cycle of the Activity [12] [13]. Resource acquisition and starting of the background services takes place in four stages namely, *onCreate, onStart, onResume, and onRestart* as shown in the activity lifecycle Figure 2 [14]. Similarly, releasing of the resources and stopping the background services performed on three phases namely, *onPause,*

onStop, and onDestroy. But most real-time applications do not follow this scenario [13], which most probably lead to performance degradation.

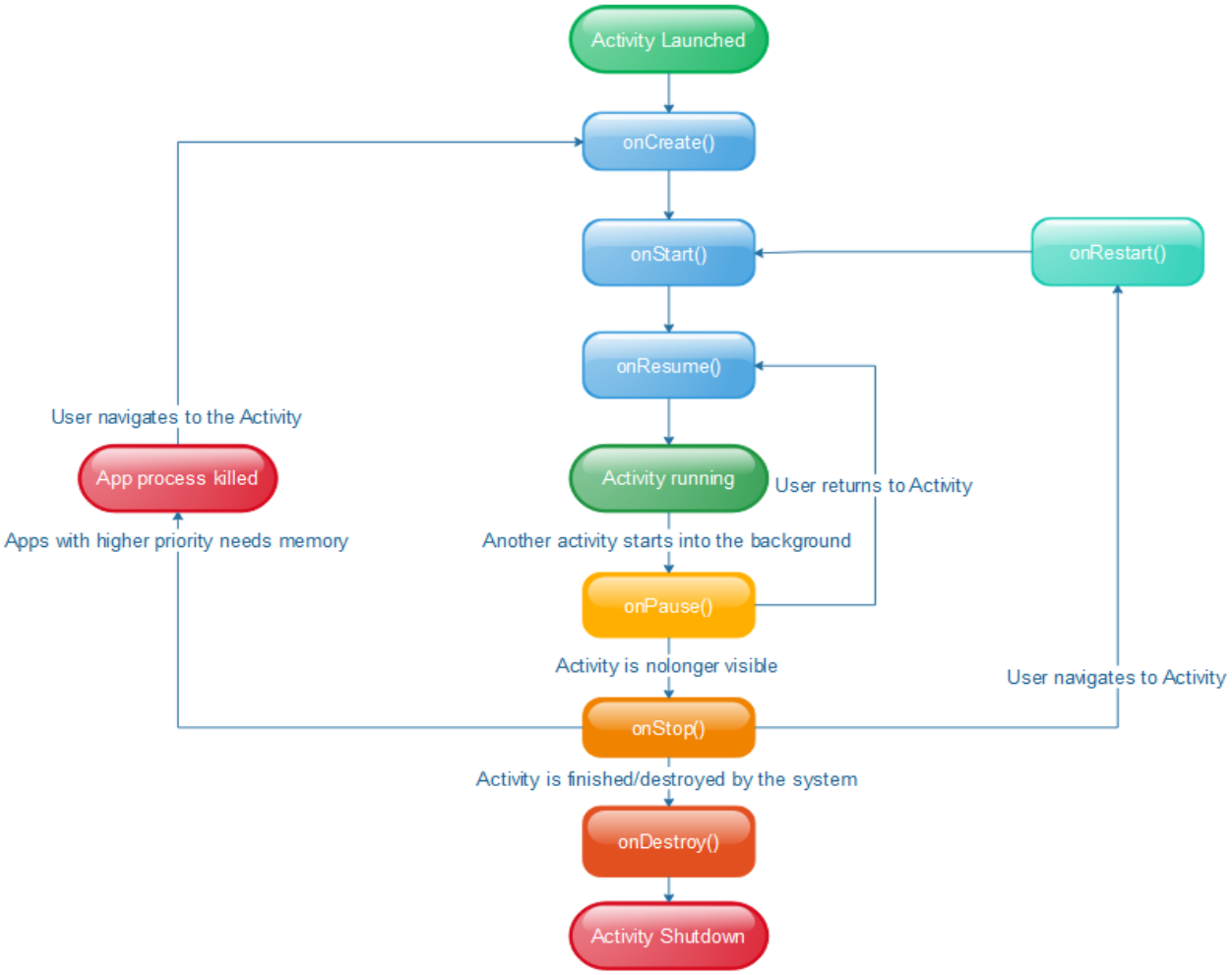


Figure 2: Activity Lifecycle Diagram

2.2 Android Memory Management

Android uses its own runtime and virtual machine to manage the memory utilization of the running applications. It verifies that the application responsiveness by stopping and killing running processes as needed to free up resources for higher priority application. The memory management is performed with the help of Dalvik Virtual Machine (DVM) and Android Runtime (ART) core libraries [15]. The difference between these two runtimes is shown in Table 1:

Table 1: Comparison between the two Android memory management environments

Dalvik Virtual Machine (DVM)	***Android Runtime (ART)***
Just in time (JIT) compilation	Ahead of Time compilation (AOT)
In each run, dynamically translates Dalvik bytecode into Machine code	During app installation, statically translates DEX bytecode into machine code and stored in the device's storage
As the execution progresses, huge bytecode is compiled and cached	A one-time event, which happens only once during app installation on the device
The app runs slower than ART	The app runs faster (improved performance)
Requires extra time for compilation	The app starts faster because no extra time is needed to compile during runtime
Works better for devices with low storage as space occupied is lesser	Consumes more internal storage space since it stores APKs and compiled apps.

2.2.1 Garbage Collection

When the managed memory management environment like Dalvik Virtual Machine and ART decides a memory is no longer being used by the running program, it tries to free it back to the heap without any need of intervention from the programmer. This mechanism for reclaiming back the unused memory resource within a managed memory environment is referred as garbage collection. The two major goals of garbage collection are searching the data objects that cannot be accessed in the program in the future and releasing (reclaiming) resources used by those data objects [15]. Every time heap generation starts to fill up, the system begins garbage collection execution events to free up the memory. The garbage collection duration to free up the heap depends on the number of active objects in each heap generation. No matter how the garbage collection is fast in its execution, it affects the performance of the application [16] [15]. When the garbage collection started in the middle of execution or intensive processing, it can increase the processing time which will lead to the application slowness by reducing the response time. On the other hand, if the processing time is increased in UI application, then the app will pass the recommended 16 ms threshold of smooth frame rendering.

Garbage collection works with basic three steps. These are traversing through all object references in a memory starting from GC roots and marking active objects which has references from GC roots (local variables in the main thread, the main thread, and static variables of the main class), Objects which are not marked are removed from the directory, and finally rearranging live objects [17].

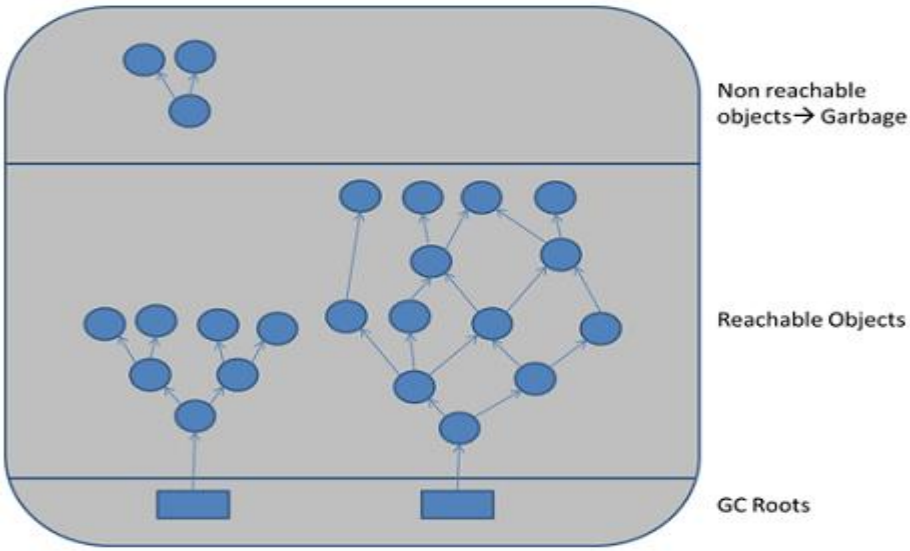


Figure 3: Mark Live Objects from GC roots

2.2.2 Allocation and Reclamation of Application Memory

Dalvik heap is limited to a single virtual memory range for each of the application process. This logical heap size can grow as required up to the limit that the system defines for each application. Android shrinks the logical heap size when unused (free) space is found at the end of the heap [15]. After every GC, Dalvik walks through the heap and searches free pages, and returns those pages to the kernel.

Android has a module for memory conservation mechanisms such as Low Memory Killer (LMK), Activity Manager Service, and Out of Memory Killer (OOM) [18] [19] [20] [16]. Despite its importance, these modules generate several kernel function calls that trigger sluggish responses and ineffective application cache utilization, which reduces the speed of app relaunching. In these modules, a required process might get killed and results increased the number of memory loading cycles as the processes go through this lifecycle to start again.

2.2.3 Memory Resource Leakage

When allocated object in running application lives longer than the life inside an activity [21] or a memory which is no longer needed is not released on time is called a memory leak. The following are some of the factors that lead to memory leakage:

- *Programming Languages:* Languages that don't have built-in automatic garbage collection like C and C++ are prone to leaks as the dynamically created memory would become unreachable.
- *Creation of Unnecessary Objects:* Allocating memory is much more expensive than not allocating. When more objects are allocated in the running app, periodic garbage collection will be forced which will endanger the user experience as it reduces the performance of the app [22].
- *Programming Styles:* The programming bugs that the programmer lefts, when it is no longer needed and has not dereferenced the code, leads to memory leak [23]. Moreover, the *Failed Invocation of Memory-Related API Calls* and Lack of *Exception Handling* [21] and bad implementation practices [24] also leads to such memory resource leakage that results in app crashes, freeze, and slowdowns.

2.2.4 Object Reference

An object reference includes addresses and class information about the object being referenced. Based on how objects are garbage collected, references to those objects in Java are classified into four types [25]: strong references, weak references, soft references, and phantom references. The possible way of releasing object references that the application holds is by making the memory available for the garbage collector.

- *Strong References:* In a programming, if an object has strong reference attached to it, then it will not be eligible for garbage collection since there are live objects that reference to the object. When an object is referenced by a chain of strong reference objects, it is not possible to perform garbage collection.
- *Weak References:* This type of object is eligible for the garbage collection when the JVM automatically runs the garbage collection.

- *Soft References:* This type of reference objects is like weak reference object that persists in memory till memory is available. Soft references cleared before JVM throws an *OutOfMemoryError*.
- *Phantom References:* Unlike soft and weak references, this reference is not automatically cleared by the garbage collector. On the other hand, it is garbage collected, but will be kept in *ReferenceQueue* before removing it from the memory. Moreover, objects reachable through phantom reference remains until references become unreachable or cleared themselves.

2.2.5 Memory Leak Pattern

In software development, a pattern is a written document that shows or describes the general solution for a design problem that occurs repeatedly in various projects. A pattern can contain the description of objects and object classes to be used with attributes and dependencies and the general approach for how to solve the problem.

On the other hand, memory leak patterns contain objects and object classes that reveal the existence of the memory leak within a program. This leak patterns can be caused by the running applications with subtle vulnerability and leads to unwanted consequences such as application crash [21]. In addition to this, keeping the references of unused memory objects such as views and drawable images for a long time leads to the denial of the new memory allocation.

2.3 Performance Analysis and Optimization of Android Applications

The mobile application performance is critical to keep the reputation and the success of the application. The application end users do not tolerate apps that do not perform well. If apps are poor, it will be a subject of uninstallation. In the previous studies [2], [21], [24], [26], [27], [28], different areas have been explored and investigated for the possible improvements of the performance of the Android applications.

Applications need to be tested automatically to explore the possible defects in its performance and optimize the test sequences. To minimize the efforts needed in testing and detection of the available defects in the program, various researchers try to address the problem by using different testing techniques. Even though, there are lots of research regarding leak detection techniques, leak

testing, and performance testing, until now there are no automated means of testing the memory performance through continuous delivery to monitor the possible changes in the source code.

To improve the quality of codes and performance of apps various mechanisms are addressed such as correction of code smells [26] [24], picture smells [24], and micro-optimization [27] by evaluating their impact on memory and CPU usage. Performance related issues are mainly found in applications user interface [26], parameters controlling hardware (e.g. GPS) [28], extensive usage of sensors and network data [2], and failure to handle specific exceptions (for acquiring and releasing of resources) [21]. While developing applications, improper management of resources leads to crashes, slowdowns, and negative user experiences [29] and energy consumption [30] as well results in a complaint which affects app's rating and the chance of success [4]. Performance of apps is analyzed in the following four categories: Memory, Energy, Network, and GUI. In this study, we are only focused on Memory issues that will affect memory performance of apps.

2.3.1 Memory Utilization

Resource leaks such as memory and battery are caused due to improper management of resources [29] and missing of releasing operations [31] which results in performance degradation, crashes, slowdowns, and negative user experience. Among the different techniques that has been studied for the detection of resource leaks are: at the bytecode level [31] by converting APK into DEX bytecode and construct FCG, testing of leaks based on GUI model specifically on coverage criteria [29] in which GUI events should have a neutral effect and should not increase the resource consumption.

Among the various techniques that has been applied to improve memory performance of Android applications, inserting patch code in to bytecode of apps [32], extensive caching or memory management using user's application usage pattern to decide for killing of apps as well as setting background cache limit [16] to improve performance while number of garbage collection calls and abnormal memory cancellation [16] [33] has a negative impact on performance. To improve such memory performance issues, reduce process termination and process triggered by LMK (Low Memory Killer) based on application state and size for apps with larger objects [34], prevents LMK and OOMK (Out of Memory) from killing application process due to inefficient memory using memory portioning at operating system level [35].

In addition to this, plenty of studies proposed various methods used for improvements of the memory performance. Heap size is limited and dependent on the devices [16] [33]. Garbage collection has a negative impact on the performance level of applications. The other studies by Lim et al. [35] assess the enhancement of memory performance by suggesting memory partitioning scheme at the operating system level targeting the avoidance of Out-of-Memory Killer and Low Memory Killer

2.3.2 Energy Consumption

Energy efficiency is the decisive factors in mobile systems that direct the recent researches towards the reduction of dissipation in mobile applications and hardware [36]. Android has a programming interface for managing and optimizing the energy consumption through acquiring and releasing functions in the application.

The possible cause of abnormal energy consumption of android applications includes different factors. Among those factors, code smells [37], energy hotspots/ bugs [13] [38], development approaches such as Java, JavaScript, C++ [30], runtime (ART Vs Dalvik), implementation choices (serial Vs parallel and recursion Vs iteration) and programming languages [30] [39] are identified in the study.

Energy leaks can be detected using POEM (Portable Open Source Energy Monitor) [40] by performing analysis on the call graphs, basic blocks, and Android API calls. The question of why instead of how much power is consumed on a device is addressed. Similarly, state-taint analysis [41], and Energy Patch [42] which uses dynamic and static analysis techniques to detect, validate, and repairing possible android application energy bugs has been stated in the study.

The impacts of code smell correction on energy consumption [37] are investigated by developing a tooled approach called HOT-PEPPER towards *Internal Getter/ Setter, Member Ignoring Method, and HashMap Usage* as used by [24], and *Picture Smells* (compression, bad picture format, and bitmap format). The study concludes the energy consumption of apps can be reduced by 4.83% while correcting code smells.

On the other hand, energy consumption apps can be reduced by killing unnecessary applications running in the background and adjusting CPU frequency, offloading computation intensive parts of apps to the server and executing the rest on smartphone devices [9].

2.3.3 Network Traffic and Performance

The performance of the android application can be optimized by reducing the size of network traffic which will lead to the reduction of battery drain. As stated by Google developers, reducing the size of the file being downloaded also helps the developer to provide a better network experience to the user.

The networking issues are also one of the factors that affect the performance of applications and that needs to be assessed and evaluated. Energy can be excessively consumed due to cellular or Wi-Fi connection when running different apps and background network traffic [43] [44]. More than 80% of apps transmit their 80% of background data in their first minutes when the app is sent to the background state. The network traffic data can be reduced by half if an operating system can terminate long-running applications [44]. On the other hand, the reduced signal strength lowers the data rate which elongates the packet transmission and that leads to increasing in energy consumption of apps [43].

2.3.4 GUI Rendering

In addition to memory leak patterns, the developed applications user interface issues such as slow rendering or skipping of frames are also the leading actor for application crashes and slow response.

The developed application UI should load faster to satisfy end users expectation. Beyond getting the contents of the screen quickly, it should render in a smooth way. The Android developer team describes the unsmooth motion as Jank, which is caused because of missing a screen frame refresh. Android devices refresh the screen 60 times per. This is because the screen is refreshed every 16 ms (i.e. 1s/60fps = 16 ms per frame), and it is essential to ensure all the rendering takes place in less than 16 ms. Whenever a frame is skipped, end users will face a skip or a jump in the animation. So, to keep the animations smooth, there should be a way to ensure that the entire screen renders in 16 ms.

User interface testing makes sure that application not only meets its functional requirement, but also the user interaction screen rendering is smooth which is running at 60 fps (frames per second). Skipped, delayed, or dropped frames is known as Jank, especially it occurs in the UI applications that have animation or scrolling nature in it. The act of generating a frame from the app and displaying it on the screen is said to be rendering.

GPU rendering and other related GUI issues are the possible cause of application crashes, slowdowns, or freezes. The study by Shin et. al. [45] identified as there is a different level of overdraws for most of the applications. As the number of overdraw increases, the performance of rendering will be reduced as it takes too long time to redraw each pixel and to build the views on the screen. On the other hand, event handling thread or main thread may perform excessive work that leads to unresponsiveness. While testing applications for responsiveness [46], the researchers identified as there is no tools or software engineering research to perform and expose the janky issues automatically in android applications.

Google Android developers [22] proposes three methods of Jank identification. These are visual inspection (to quickly run through all use cases in the app in few minutes), Systrace (to provide more detail but running all use cases would be difficult as it would be flooded with plenty of data), and custom monitoring performance (to measure specific part of an app on devices running in the field).

2.4 Related Works

Automated testing is effective and efficient as it saves time and cost of the developer. However, most apps are being tested manually [47] [27]. Based on the surveys conducted [47] [27], application developers use manual testing over automated testing due to the factor of test case representation of the activity and the cumbersome natures of the automated testing tools [47]. However, in this study, the simplified automation test is developed to test the memory consumption of apps. But, the test cases used in the automation testing is not valid for all type of testing as it needs to be modified based on the nature of the applications and the type of activities to be performed on the device.

Resource leaks can be tested using systematic testing which is based on GUI model. The study by Yan et.al. [29] conducted systematic testing by proposing a novel approach which considers a GU

model based on neutral cycles. The study assumes that GUI events have neutral effect and doesn't lead to the increasement of resource usage. But, the study did not consider the impacts of codes that allocates and reclaims important resources

Resource leaks such as memory leaks can be detected using fuzz testing [21] and code patterns [48]. In the first case, leaks can be detected by inputting huge amounts of data and making an application crash. The study suggests considering extra memory leak patterns such as inner class, context, and database connection leaks. In the latter case, detection takes place using code patterns at an object level depend on object references. Parts of codes responsible for the leak are identified. But, in this study, investigation takes place using resource leak patterns at source code level by understanding the memory features of the source code and optimizing the source code. The reason for choosing the source code level optimization is the nature of the source code. Because it is the interface between the developer and the hardware resources such as memory. In addition to this, the issues that are not addressed such as leaks due to event-based memory leaks, thread activities, inner class, context, and static reference will be evaluated.

In addition to the testing and detection of resource leaks, the study by Liu et. al. [32] proposes the possibility of fixing bugs using static analysis and code instrumentation by inserting patch codes at the byte-code level. The study uses Relda2, a static analysis tool which reports the possible code defects. The reports generated by this analysis tool is used as an input for starting the fixing process. Apart from the problems solved, it doesn't consider fixing of the resource leak without introducing new bugs.

On the other hand, Hecht et.al. [24] assesses positive performance impacts of android code smell corrections. The study focuses on performance code smells namely; *Member Ignoring Method, Internal Getter/ Setter, and HashMap Usage* using evaluation metrics such as a *number of delayed frames, frame time, and number of garbage collection calls.* As the study indicates there is up to 12.4% improvements in UI metrics and 3.6% on memory-related metrics while correcting the abovementioned android code smells.

Moreover, Lim et. al. [35] proposes a memory partitioning framework at the OS level. The study provides the capability of limiting memory consumption of non-trusted applications and the avoidance of LMK and OOMK operations on physical memory shortage. But, the study doesn't

address the issues that triggers the LMK and OOMK. In addition to this, there is no algorithm for the classification of trusted and non-trusted applications.

Table 2: Summary of Related Work

Ref No	*Open issues*	*Problem solved*	*Not yet covered*	*Will be addressed by this thesis*
[21]	Automatic Conversion of leak patterns into test cases to detect memory leak vulnerability early	Proposed fuzz testing for the detection of memory leak	Memory leak patterns such as context, inner class, and database connection leaks	Context leak and Inner class will be addressed
[24]	Automation of code smell correction	Assess the positive performance impacts of correcting code smells such as MIM, HMU, and IGS	Doesn't consider ART runtime	ART will be considered
[29]	Improved resource management through new software abstraction and patterns	Proposes a novel approach for testing of leaks in GUI model	Analysis of impacts of codes that allocates and reclaims important resources	Analysis of codes will be performed
[32]	Instrumentation overhead (lines of code while patching in to byte code)	Fixes specific bugs using static analysis and code instrumentation by inserting patch codes at the byte-code level	Fix the leak without introducing new bugs	Leaks will be fixed without introducing new bugs

[35]	Consideration of memory conserving modules of Android OS	Proposed virtual memory node techniques for memory partitioning at the OS level	Doesn't consider issues that triggers the LMK and OOMK	Analysis of Causes and impacts of LMK and OOMK will be addressed
[48]	Library of classes to detect more classes of code patterns such as memory leak due to large data, multithreaded activities and event-based memory leaks	Proposes a method for identification of memory leak vulnerabilities during development and testing through identification of certain uses of code or code patterns in the application	Thread leak, Handler leak, AsyncTask activity leak, and static object references	Thread leak, Handler leak, AsyncTask activity leak, and static object references will be considered

As described in Table 2, memory leak pattern such as context leak, inner class leak, thread leak, handler leak, static object reference leak and AsyncTask leak will be thoroughly studied and the best possible optimization will be applied to improve the memory utilization of the apps. Moreover, the test will be performed on both DVM and ART runtimes as they have different memory management capabilities.

CHAPTER THREE

METHODOLOGY OF THE STUDY

This section addresses how the data is collected, organized, implemented, analyzed, and interpreted for the study. In addition to this, detail description of each method of the study is stated in Figure 4.

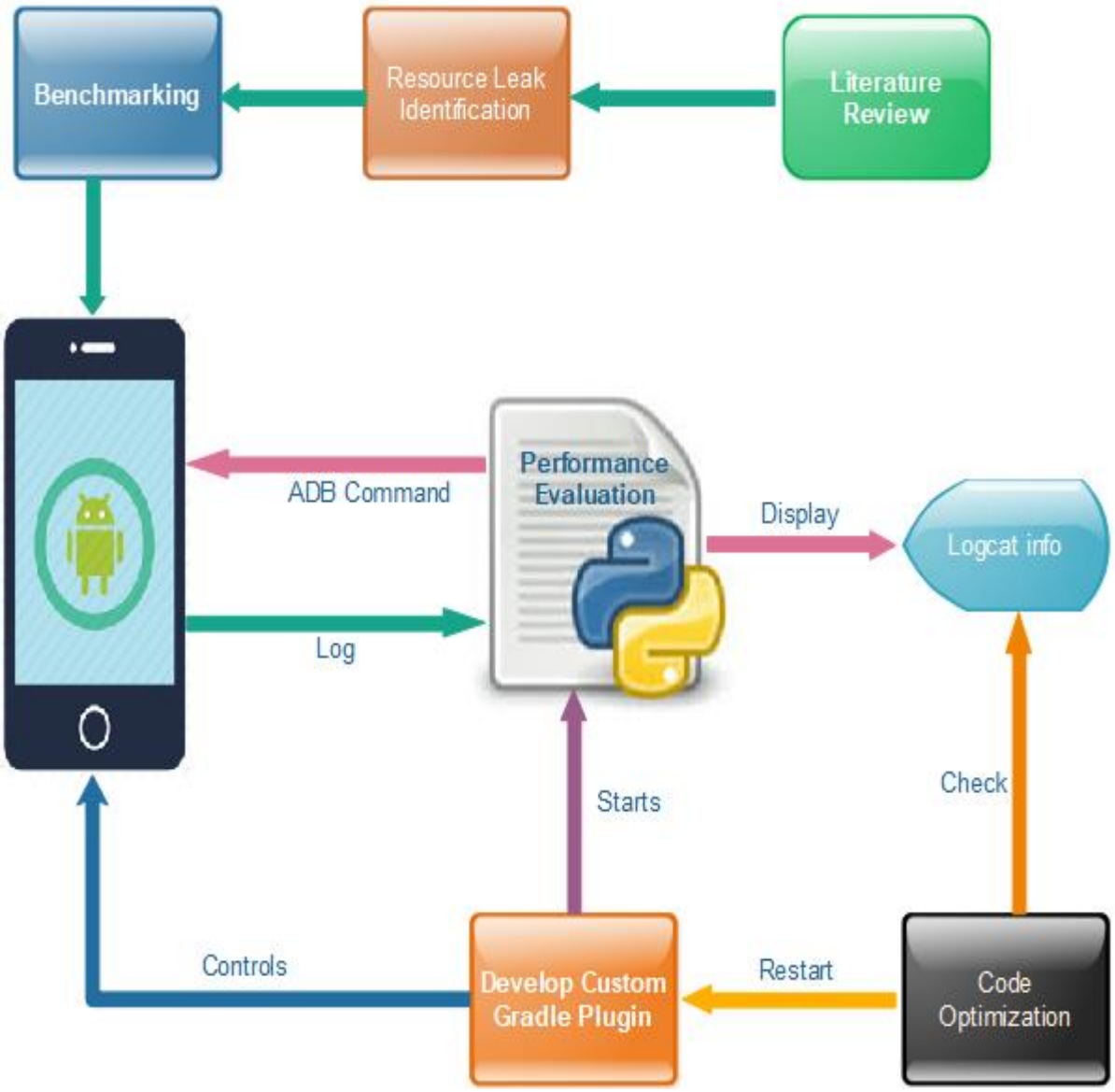


Figure 4: The Overall Procedure of the Study

3.1 Literature Review

A literature review has been conducted to assess concepts, tools, techniques, and applications of performance analysis and to get domain knowledge about the problem. To get a thorough understanding of how memory is performing within smartphones and how programming practices pave the way for efficient utilization of these resources manuals, relevant documents, and android website were reviewed.

The literature review in this study has gone through five steps:

- First, the research questions that fascinates this study is identified and defined.
- Then, the different set of keywords that enables us to find the largest possible set of publication within the scope of the study is itemized.
- To limit this study to some very relevant publications, the exclusion criteria (publication should be since 2012 and must be related to android memory consumption, leak pattern, and automation test) was applied on the search results, thus the possible papers that are not relevant to the study are excluded.
- Then after, additional keywords are itemized from the filtered papers. At this stage, keywords are used to categorize the possible studies on the addressed problems contribution and focus areas of the researchers.
- Finally, the data is extracted and mapped to the appropriate sub-categories which are created for this purpose.

3.2 Search Datasets

Our data search is based on the repositories, which help us to find relevant publications. To find data sets that are relevant to the study, first select four well-known electronic repositories are selected; namely, ACM Digital Library, IEEE Xplore Digital Library, arXiv, and Science Direct. But, in addition to identified repositories, research papers are used in different publications as needed.

3.3 Search Strategy

Studies need to be directly related to android application focusing on resource leak, performance enhancement, memory performance testing techniques, challenges, methods or approaches.

The strategy is performed using the following aspects:

- Search for synonyms and alternate words.
- Use Boolean OR to associate alternate spellings and synonyms.
- Use Boolean AND to combine major terms together.

For each try, the search string is evaluated based on how the returned result is relevant for this study area i.e. android application memory performance analysis or testing specifically for memory resource utilization.

Based on our initial research review, 12 papers are identified as a reference to evaluate the effectiveness of search strings to retrieve those studies. The search strings which retrieves the most relevant papers to our study and returned the maximum number of the previously known or selected studies will be chosen. The strings which retrieves less and less relevant to our study will be excluded from the list.

Table 3: Search Strings applied on IEEExplore

No of try	Search String	No of studies missed	Returned results
Try 1	((android) AND (application OR app) AND (memory) AND (performance OR "resource leak") AND (test OR analysis))	4	28
Try 2	(("android app" OR "android application") AND (memory) AND (performance OR "resource leak"))	9	74
Try 3	((android) AND (application OR app) AND (memory) AND (performance) AND (test OR analysis OR "automated test"))	10	44
Try 4	((android) AND (application OR app) AND (memory OR automated testing OR battery OR energy) AND (performance))	8	22
Try 5	((android) OR (performance) OR ("resource leak") OR (memory) OR (energy))	0	691

On the other hand, several studies published using the term “app” instead of “application”. “energy” also used in combination with memory. Hence, we incorporated it in our search string. This needs to be done carefully in order not to struggle with the bulkiness of the results retrieved.

3.4 Study Selection Criteria

Studies that are taken into consideration are only the studies which is supported by empirical evaluation or data i.e. if a research paper proposes a new method, testing approach or programming tips, it should show some sort of evidence that supports the proposed approach for its effectiveness.

In the systematic literature review used in this study the following inclusion criteria are used:

- Studies must be related to android application performance testing specifically memory utilization or enhancement, memory consumption or allocation challenges, and limitations.
- Studies must be supported with empirical data.

To exclude the irrelevant papers the following exclusion criteria are used:

- Non-English publications are filtered and excluded.
- Because our search terms include “performance” and “energy” to collect most papers, the collected set includes papers about “hardware device performance”, “android OS performance”, and other fields energy, such non-android application related papers are excluded.
- Papers that present the opinion without any supporting empirical evidence.

3.5 Study Selection Process

The search process is applied on all databases that has been previously identified. The initial year of publication were specified to 2012 during search and restricted to studies related to computer science. The search for literature is covered up until 2017.

The paper selection process was iterative and incremental. Hence, it passes through different phases. Initially, the database is searched using search string. then after, in the first phase, the retrieved paper is filtered based on their title and abstract. In this step, papers that are not related to android application memory utilization, performance, testing, and enhancement are excluded from the list.

In the second phase, papers are filtered by applying selection criteria through reading introduction, methodology, and conclusion. In this step papers may be excluded because it is not empirical or did not confirm to study selection criteria. Final step involves complete and thorough reading of a paper.

3.6 Classification Scheme (abstract keywording)

The keywording process consisted of two phases which is applied to the final selected papers.

Firstly, the researcher reads abstract of selected papers and searches for keywords that shows the findings or results of the study. When abstracts are written with poor quality or too short for choosing significant keywords, researcher reads introduction and conclusion as well.

Secondly, the set of keywords identified from different retrieved papers combined to understand the context and contribution of the paper. This provides a capability to identify sub-categories for classification scheme.

3.7 Resource Leak Pattern Identification

After a thorough investigation, plenty of leak patterns are found which are then grouped into two categories: long running task, and static reference.

- *Long Running Task:* This is a category of resource leak pattern of an activity that runs for longer time. It includes AsyncTask Activity, Handler Activity, Leak Threads Activity, and Runnable Thread leaks. Long-running tasks or worker threads live longer than the activity.
- *Static Reference:* This is a category that leads to resource leak due to static references. It consists of Static Inner Class Activity, Static Variable Activity, Static View Activity, and Singleton Activity.

3.8 Bench Marking

The sample application that contains the identified leak patterns is created for evaluation of its activity for later reference that helps to evaluate the newly suggested approaches performance or effectiveness.

3.9 Develop Custom Gradle Plugin

For every device that is connected to the current working application, whether it is an emulator or real device, this custom Gradle helps to automate the creation of performance testing and analysis. In addition to this, it provides the functionality to execute performance tests without leaving the android studio and presents Gradle task for continuous delivery to perform local android device tests using monkeyrunner.

3.9.1 Application Testing

The benchmarking application is tested with the help of the custom Gradle plugin tasks. The developed and existed leak patterns apps need to be tested at the functional/framework level. Moreover, the test data on the emulator and real devices need to be controlled to extract test data to the external directory. This operation is performed using monkeyrunner tool. The monkeyrunner helps to start the python script for performance evaluation. It also allows to start and execute the test suite in parallel with threads in the running program.

3.9.2 Performance Evaluation

It is a python script which enables to test android apps installed on android device using ADB command and retrieves the log files such as logcat info (to display number of GC calls and the general information of activity lifecycle and memory management process of ART and Dalvik runtime) and moves this result to the external data directory for the simplicity of the analysis process.

3.9.3 Instrumentation Testing

This is the test which require the android system and that have accesses to instrumentation information, such as the context of applications under test. While using instrumentation testing, in the background, apps will be installed which will control an app, launching it and running UI tests or non-UI logic tests as needed. Instrumentation tests access android components such as clicking button and application lifecycle. This testing is coordinated by the developed custom Gradle methods with the help of monkey runner to control the device and starting of the performance evaluation script for analyzing the memory consumption and to enable the test data to be retrieved from the debuggable test APK to the external storage directories using ADB commands.

3.10 Code Optimization

The possible refactoring or restructuring of codes are in place without altering the external functionality of the benchmarking app based on the outputs which are pulled from an android device with the help of python script. After all, the optimized code is written for the existing resource leak patterns to be tested. The later test result is used to compare the output with the previous one for a possible improvement in performance. If the problem is not fixed, a way of optimization will be done in a continuous way.

CHAPTER FOUR

HIGH-LEVEL ARCHITECTURE OF THE PROPOSED SOLUTION

In this section, high-level architecture of the proposed solution is presented in detail to show each component that exists within the proposed solution. The identified resource leak patterns are developed in a way that developers are trying to code which leads to an abnormal resource consumption and degrades the performance of developed applications. As it is depicted in Figure 5, each component of the solution is presented, and detail descriptions are provided. The components within the architecture and the relationship between components are identified based on the functionality they provide and the objective of the study stated early in chapter one. The architecure is also developed from the scratch.

4.1Resource Leak Patterns

This is a component which shows the patterns that leads to abnormal resource consumptions and performance degradations. These leak patterns are sub-grouped into two categories for the purpose of this study as follows:

- *Long Running Activity:* this leak category is about the patterns that hold a reference for a long time even after the activity is destroyed. There are three sub-categories: AsyncTask Activity, Handler Activity, and Thread Activity.
- *Static Reference:* Static reference can live as long as an app is in memory. An activity has its own lifecycle which is usually destroyed and re-created multiple times during applications lifecycle. If an activity is referenced directly or indirectly from a static reference, the activity will not be garbage collected after it has been destroyed. This category is intending to assess such effects on performance and resource usage of an app.

Optimized Code

The outputs which are the result of the re-arrangement or refactoring of the source codes without altering the functionality of the application. Based on the test results the possible causes are identified and traced to be fixed for another round test to check whether the problem is solved, and performance of apps are optimized.

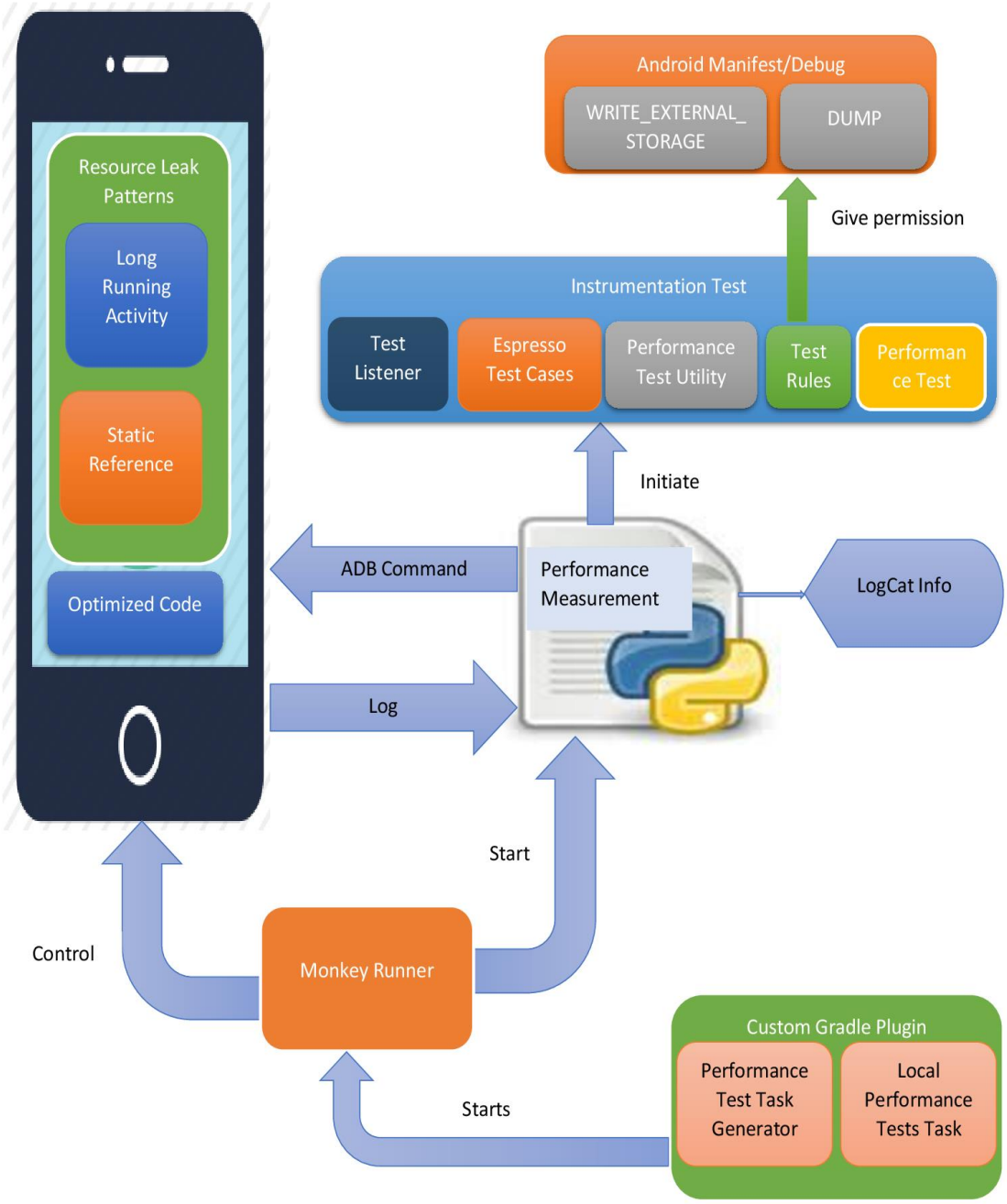


Figure 5: Architecture of Proposed Solution

4.2 Performance Measurement

The performance analysis script helps to organize the running android performance analysis and enables to collect memory information used by running the application in an automated way. It also enables storage and dumps permission in which the results found on running device or emulator to be extracted in the external storage directory for the simplicity of the analysis. If there are previously done analysis or tests on the outputs directory, it will clean it up for the new tests. In addition to this, it helps to define the performance test parameters that should run before the test is executed. The collected results with the help of this python script include logcat info (number of garbage collection calls and the general information of activity lifecycle and memory management process of ART and Dalvik runtime) in the external directory to the device.

4.3 Custom Gradle Plugin

It is a section that automates the creation of performance tests, by checking for changes whenever there is an alteration or execution of the code in the continuous delivery. Custom Gradle plugin consists of two components: Performance Test Task Generator Plugin and Local Performance Tests Task. The first one is used to automate the creation of performance testing tasks for every device that is currently connected to the currently running system. This component can be used in Gradle files (build.gradle) by calling it using “apply”. Performance Test Task Generator Plugin defines the activity that should run after completing performance test tasks, creates parent performance test task that can execute all the device specific tasks, creates performance test tasks for each connected device to the system, verify each device-dependent (specific) tasks run by parent performance test task, and verify post-test tasks run after the performance test tasks.

The later one is a Gradle task enables to execute local android device tests with monkeyrunner. It provides a capability of automated performance analysis without leaving Android Studio. In this component, the monkeyrunner and performance analysis script path is defined.

4.4 Monkey Runner

Grants permission using ADB commands for enabling to write to the external storage and for tracing performance issues. It also allows to start and execute the test suite in parallel with threads in the running program.

4.5 Instrumentation Test

This component accesses instrumentation information, such as the context of applications under test and Android components such as clicking button and application lifecycle which is intended to show the effects of each pattern on performance. It has major five components which facilitate the performance test of an app: Test Listener, Espresso Test Cases, Performance Test Utility, Test Rules, and Performance Test (annotation).

- *Test listener:* helps to initiate pre-test and post-test procedures which includes cleaning and moving files from the internal application data directory to the external one. It is also used to indicate whether there is a fatal test exception such as OutOfMemoryException. In addition to this, it enables log files to be pulled with memory information. It also checks whether test run is complete, successfulness of the extraction of files for the test, and moves files from apps internal file location to external location with ADB pull commands.
- *Performance Testing Utility:* a centralized method to ensure tests perform certain actions in the same way. It also retrieves the directory to where test files should be written to. This directory is local and removed when an app is uninstalled.
- *Performance Test:* used to annotate a class and methods that contains a performance component.
- *Espresso Test Cases:* these are test cases for resource leak patterns that are initiated by espresso view matcher by clicking the button on each leak patterns.
- *Test Rules:* these are rules that enable to execute tests and records the output of the test results at the end of the test.

4.6 Android Manifest/Debug

Android Manifest/Debug: the permission which is granted by monkeyrunner should be defined in the manifest file writing to the external storage or directory and dumping information for analysis purpose.

CHAPTER FIVE

IMPLEMENTATION OF THE PROTOTYPE

In this section the detailed implementations are described with screen captures that shows the performance test and analysis results for further improvement. The experiment of the prototype is carried on the following configuration of emulators and real-time smartphone devices. The prototype application and the optimized modules are deployed on the implementation devices as system application with the permission to read memory dumps and the memory consumption alongside with the object reference associated to the leak or the objects that causes a leak and the activity callback where the leak starts are dumped to the external storage directory for the simplicity of the analysis.

5.1 Implementation Setup

Before evaluating the memory consumption of the memory resource leak patterns and performing the automated tests, the tools and frameworks are identified based on some criteria. The Android Studio development environment is selected based on six selection criteria [49].

Table 4 : Comparison of Eclipse and Android Studio IDE

Selection criteria	***Eclipse***	***Android Studio***
User Interface	Complicated	Simple and easy
Build Systems	Apache Ant (XML based and comes with via a plugin)	Gradle build system with Groovy DSL, less verbose and easier to write and read
Code Completion Feature	High-end java autocompletion	Better autocompletion due to indepth IntelliJ IDEA support which makes the code completion less prone to error
System requirements and stability	The higher amount of RAM and high CPU speed and less stable	Slow system requirement and more stable

Speed factor	Takes 2-3 minutes for building a final release	Takes less than 40 seconds
App testing and debugging	Is not easy to test	Have tools for easy app testing

In this study espresso testing library which provides Espresso API for automating testing tasks such as writing user interface test to simulate user interactions within an app, match the button to be displayed, perform an action, finally assert the action is properly performed and a monkeyrunner for starting the script, controlling the device outside of the code, testing applications and devices at the functional or framework level and automating the test are selected. The matrix used for selection is used as illustrated in [50] is also applied in this study.

The reason for using UI frameworks on memory testing is that there is no testing framework for memory testing that enables developers to write the test cases and execute to check certain leakage in the running apps. Moreover, the developed memory leak patterns behavior should be evaluated in its activity lifecycle. This evaluation can be supported by espresso framework with simulating user interactions to the button UI that are intended to expose the leaking nature of the leak patterns. In addition to this, to simplify the analysis of the test result, in parallel with the UI interaction, monkeyrunner controls the device and performs a test on the applications at the functional/framework level.

Table 5: UI frameworks Selection for the Automated Testing platform

Matrix	***Calabash***	***Espresso***	***UI Automator***	***Monkeyrunner***
Availability	Opensource	Opensource	Opensource	Opensource
Platform	iOS and Android	Android	Android	Android
Possible operations to be performed	Gestures, assertions, and screenshots	Matching, Actions, and Assertions	Cross-app functional testing test multiple apps	Control devices and emulators outside the code
Access to code	Yes	Yes	No	Yes

Applicability	Behavior-driven development	Simulating user interaction	Change device rotation, press Back, Home, or Menu button	Test applications and devices at functional/framework level

Notepad++: the performance analysis script which is Jython in nature, is written using Notepad++ and the memory consumption and performance is evaluated by size of the leak a pattern retains and number of garbage collection calls throughout the activity life cycle. The reason for snubbing python IDE for development is it misses multiple modules that are required for executing the analysis.

5.2 Implementation Device and Target Android Selection Criteria

The device used for the implementation is both emulator and device to see if there is a variation in memory consumption or memory performance. On the other hand, the target android is selected based on the popularity and number of downloads on Google play ecosystem [51]. In the study, even though it is not popular, the managed memory environment (Dalvik runtime) is also considered since it is necessary to see the differences in ART and DVM.

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.4%
4.1.x	Jelly Bean	16	1.5%
4.2.x		17	2.2%
4.3		18	0.6%
4.4	KitKat	19	10.3%
5.0	Lollipop	21	4.8%
5.1		22	17.6%
6.0	Marshmallow	23	25.5%
7.0	Nougat	24	22.9%
7.1		25	8.2%
8.0	Oreo	26	4.9%
8.1		27	0.8%

Figure 6: Target Android Distribution for Selection

The implementation of this study is evaluated using different Android and SDK versions. The statistics shown in Figure 6 is the statistics conducted by Android developers to show the relative number of devices running a given version of the Android platform. Based on the popularity, Nougat and Marshmallow are used to address the issues of multiple users. On the other hand, KitKat and Jelly Bean are used to consider Dalvik Virtual Machine behavior while running the memory leak patterns.

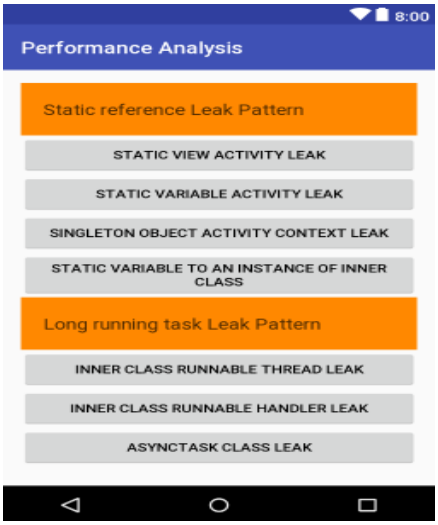


Figure 7: Resource Leak Patterns

The UI depicted in Figure 7 is the benchmarking apps GUI used for testing of the memory leak. The test cases written using espresso framework performs clicking operation (on behalf of the user) on the buttons. These buttons call the associated memory leak patterns activity. As a result, each of the test cases starts memory testing by clicking a button. Since the callbacks within the activity of the leak patterns are the major places for the memory leak, it is analyzed using the size of the leak it retains and the object reference that exists after GC performs resource reclamation.

The flowchart indicated in Figure 8 is the overall procedures followed to perform the performance test on developed apps. It has two ways of testing: specific leak pattern memory consumption using automated test and leak patterns effect of memory consumption on the running apps. In the first case, whenever the performance test finishes its execution, the test data in the logcat file is written to the external directory with associated emulator/device id, package name, test classes, and test cases. The output test file contains the detail description of each pattern throughout the activity lifecycle. The test data also contains the test failures associated with test classes (if any).

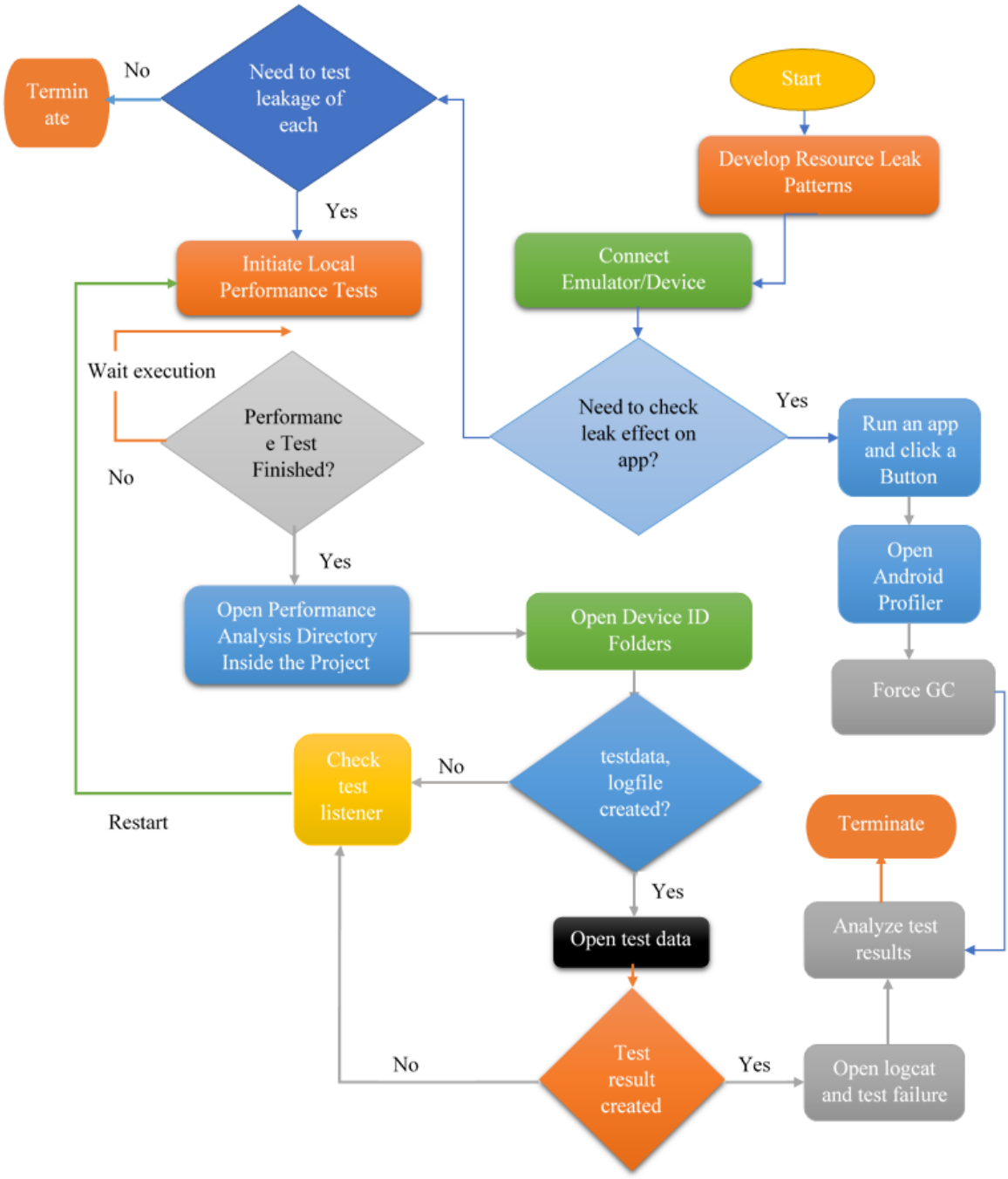


Figure 8: General Performance Analysis Flowchart

In the second case, instead of starting local performance test, Android profiler is used to cause a forced garbage collection after clicking the associated button to the leak pattern.

5.2 Resource Leak Patterns

In this section, the list of all identified resource leak patterns which leads to performance degradations is developed. There are seven developed memory leak patterns with two categories: Leakage through long-running activities and leakage through static object references. The first category consists of three leak patterns namely: AsyncTask activity leak, Thread activity leak, and Handler activity leak. On the other hand, the leakage through static object reference consists four leak pattern components: Static variable to an instance of the inner class leak, static variable activity leak, static view activity leak, and singleton object activity leak.

5.3 Resource Leakage Through Long-Running Activities

5.3.1 AsyncTask Activity Leak

AsyncTask classes are used to implement asynchronous, long-running task on a worker thread (any thread which is not the main or UI thread). It provides a capability to perform background operations and publish the output on the UI thread without manipulating handlers or threads. The execution of AsyncTask passes through four steps as depicted in Figure 10.

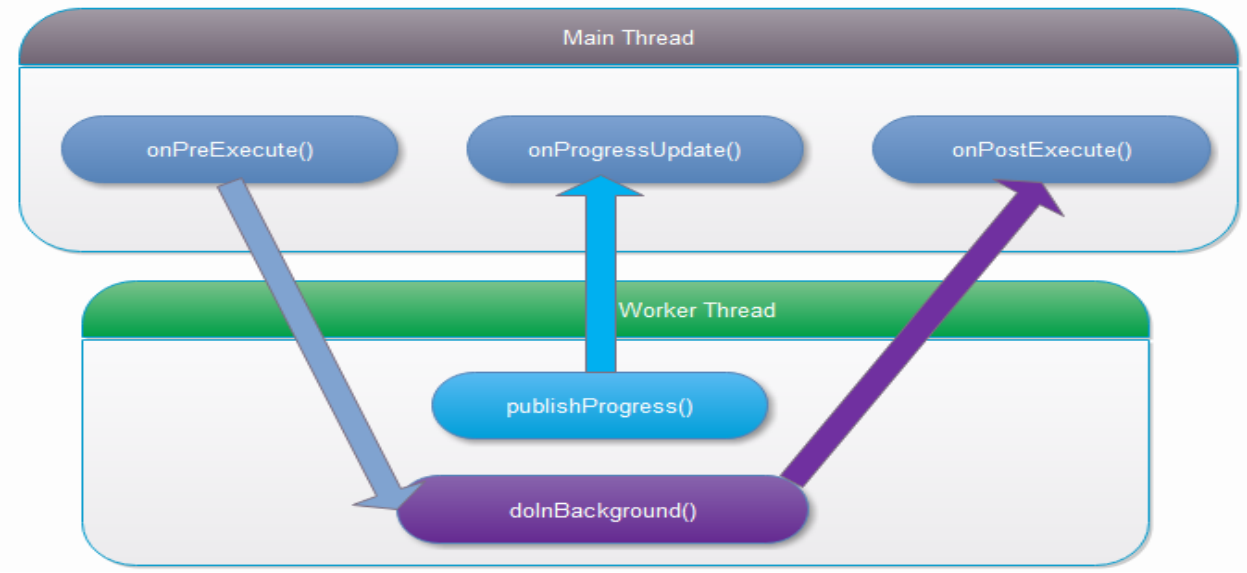


Figure 9: Lifecycle of AsyncTask Activity

The parameters in AsyncTask are: “params”, “progress”, and “Result”. “params” specifies the type of parameters passed to *doInBackground ()* as an array. “progress” indicates the type of parameters

passed to *onProgressUpdate ().* "Result" specifies type of parameters that *doInBackground ()* returns. It is automatically passed to *onPostExecute ()* on the main thread.

```
public class AsyncTaskActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.async_task_activity_leak);

        new AsyncTask<Void, Void, Void>() {
            @Override
            protected Void doInBackground(Void... params) {
                while (true) {
                    SystemClock.sleep( ms: 1000);
                }
            }
        }.execute();
    }
}
```

Figure 10: Leak to AsyncTask Activity

As indicated in Figure 10, the declaration is depicted in a manner that "params" doesn't take any parameters or it is void, "Progress" parameter type is `Void`, which means that `AsyncTask` won't use the `publishProgress ()` or `onProgressUpdate ()` methods, and "Result" parameter is void so that AsyncTask doesn't return in *doInBackground ()* which would have been passed to *onPostExecute ().* Moreover, the background task keeps an implicit reference to the AsyncTask activity while running on another thread which doesn't have a cancellation policy (2), the activity will remain in memory with all its resources attached till the background thread has terminated.

The code depicted in Figure 10 has a memory leakage when the associated activity UI button is clicked or rotated within one second after it is created. The activity is already destroyed while clicking or screen rotation. Since AsyncTask is declared as non-static (1), the background task is still holding a reference to the activity which makes the activity non-eligible for garbage collection.

The *AsyncTask* class (1) has the strong reference to the *AsyncTaskActivity*. If an activity performs background work after the Activity has been terminated, the strong reference to the Activity will persist and it won't be garbage collected till the background task finishes. The background task takes longer time. The AsyncTask object has not finished its operation and thus not to be garbage

collected. In addition to this, it has an implicit reference to the finishing activity (i.e. AsyncTaskActivity) which prevents the memory from being released.

5.3.2 Refactored AsyncTask Activity Leak

The AsyncTask program in Figure 9 has an implicit reference to (*AsyncTaskActivity*), which is the enclosing activity. When a user clicks a button or rotates the screen, the device configuration change will be triggered that leads to the termination of the activity that started the AsyncTask. But, it will not be collected by GC till the AsyncTask finishes its execution. To address this issue, the study focuses on whether the activity is in memory when it is terminated by the framework.

To avoid AsyncTask activity memory leaks, the class should be static (2) instead of non-static inner class or weak reference to reference the outer class and to access its private members (5) if any. This static class (2) doesn't have a reference to the containing activity class. In addition to this, the task should also be canceled (4) in *onDestroy ()* inside an activity.

```
public class AsyncTaskActivity extends AppCompatActivity {
5   private SampleTask sampleTask = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.async_task_activity_leak);

        sampleTask = new SampleTask();
        sampleTask.execute();
    }

    @Override
4   protected void onDestroy() {
        super.onDestroy();
         sampleTask.cancel( mayInterruptIfRunning: true);
    }
2   private static class SampleTask extends AsyncTask<Void, Void, Void> {
        @Override
        protected Void doInBackground(Void... params) {
            // should check if cancelled before next loop
1           while (!isCancelled()) {
                SystemClock.sleep( ms: 1000);
            }
3           return null;
        }
    }
}
```

Figure 11: AsyncTask Optimization by making class static and cancel the task on destroy

Instead of continuing the background task, the loop (1) should consider every one second if an object that needs to be GCd is collected. In addition to this, the memory leak can be removed using the weak reference.

```
public class AsyncTaskActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.async_task_activity_leak);
            new SampleTask( activity: this).execute();
    }

    private static class SampleTask extends AsyncTask<Void, Void, Void> {
        private WeakReference<AsyncTaskActivity> mRef;

        public SampleTask(AsyncTaskActivity activity) {
            mRef = new WeakReference<AsyncTaskActivity>(activity);
        }

        @Override
        protected Void doInBackground(Void... params) {
            try {
                Thread.sleep( millis: 1000 * 10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return null;
        }
    }
}
```

Figure 12: AsyncTask optimization by using a weak reference

As indicated in the code, by creating a weak reference of the AsyncTaskActivity, the memory leak is removed. When activity is destroyed through rotation (which results in configuration changes) or button clicking, AsyncTask is holding its weak reference so that it can be garbage collected.

5.3.3 Thread Activity Leak

For every created new activity, the thread can be leaked and can never be reclaimed. In Java, threads are GC roots, in which Dalvik Virtual Machine (DVM) keeps a reference to all active threads in the running system. Hence, since it is left running, it will never be garbage collected.

```
public class ThreadActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.runnable_thread_activity_leak);

    1   new MyThread().start();
    }

    private class MyThread extends Thread {
        @Override
        public void run() {
    2       while (true) {
                SystemClock.sleep( ms: 1000);
            }
        }
    }
}
```

Figure 13: Runnable thread activity leak

In the code snippet above, (2) *SystemClock. Sleep ()* alone couldn't cause or produces memory leaks when it is called within the method *run ()* of runnable. Rather a leak produced when there is

an object that exists in a heap memory which is not referenced from any of running thread codes. Hence, if *SystemClock. Sleep ()* doesn't have any reference capable of leaking objects, then it is not a threat for a memory leak. On the other hand, Runnable is a simple interface which possesses *run ()* method. As shown in Figure 13, the pattern contains three thread status which is (1) *New, Runnable, and TIMED_WAITING*. *New (1)* is a status in which a thread is created but has not been processed yet. In *Runnable* case, a thread is holding CPU and processing task (*Waiting status*). A *Runnable* can be submitted to a *Thread or Thread pool* to execute in a separate thread. A thread execution is shown in Figure 14 [52].

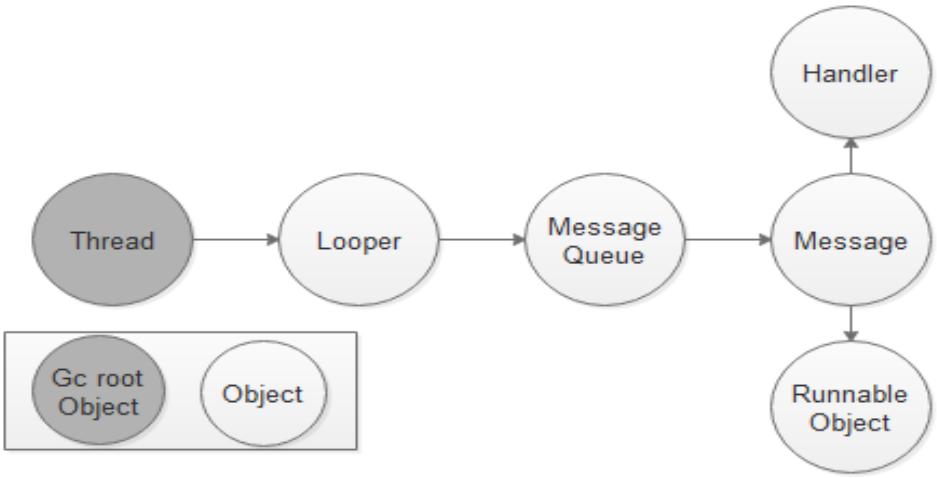


Figure 14: A thread that can receive messages, and the referenced objects

In *TIMED_WAITING, the* thread is waiting by using a *sleep* method. The maximum waiting time here is specified by method parameters. Handler instances and the objects it references cannot be deallocated until the thread terminates its execution.

5.3.4 Refactored Thread Activity Leak

The Dalvik Virtual Machin (DVM) keeps strong references to all active threads in the runtime system. So that threads that are still running are not eligible for garbage collection. As a result, there is a need for a cancelation policy for the background threads. In addition to this, the thread activity should be declared static to prevent the implicit reference to the containing activity class.

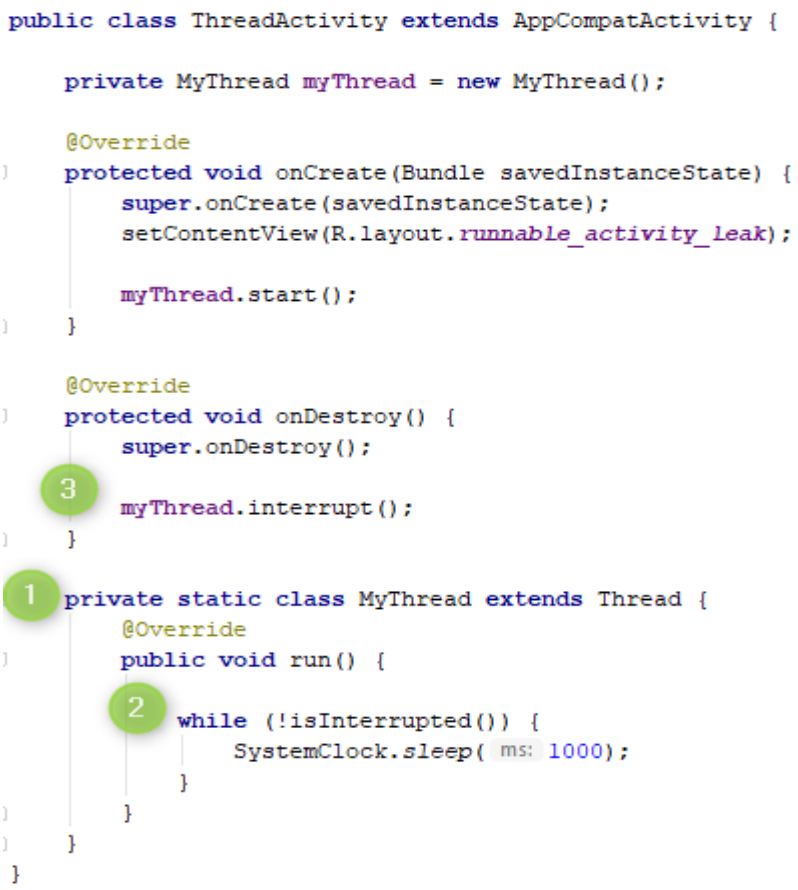


Figure 15: Thread Activity Leak Optimization

From the figure, (1) the thread class is made static, so now it doesn't have a reference to the enclosing activity class and its private members. (2) before continuing the next loop for execution, change in a configuration through rotation or button clicking and interruption (existence of cancelation) is checked. (3) if there is a configuration change or any interruption, the thread is not allowed to persist rather it will be canceled. Now, if DVM calls for garbage collection, there is no reference within an activity so that the unreferenced object will be easily collected. In doing so, the leak is removed.

5.3.5 Handler Activity Leak

Similarly, as indicated in Figure 14, when an android application starts, a *Looper* object is created for the applications main thread. *Looper* used for implementing a simple message queue and processing *Message* objects in a loop sequentially. Application framework events such as button clicks and activity lifecycle method calls comprised of *Message* objects, that are added to *Looper's*

message queue and are processed separately. The main (UI) thread's *Looper* exists throughout the entire application lifecycle.

Handler is the message processor on the worker thread. The Message is data or work (runnable) offloaded from, for example, UI thread. Whenever a message is received on Handler, it appends the message to the Message Queue. Each thread can have only one Message Queue. Looper is a message dispatcher that takes one message at a time from the Message Queue and dispatch it to the Handler for processing.

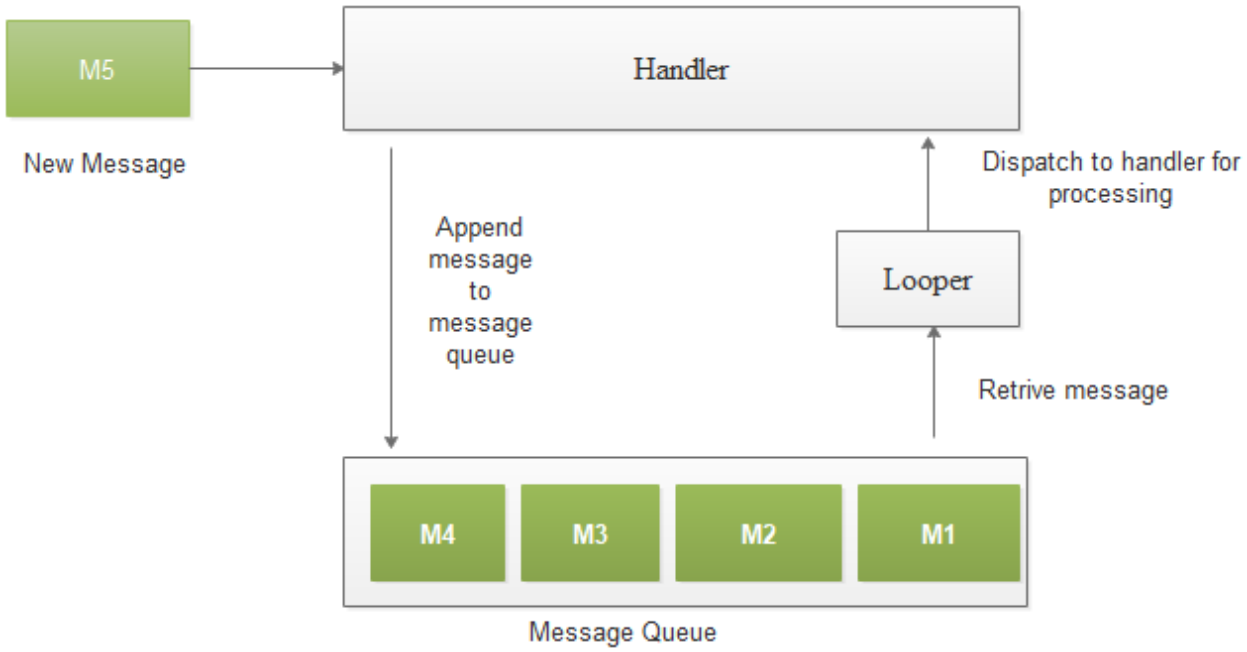


Figure 16: Message Passing in Android

As indicated in code snippet (Figure 17), when a *Handler* is instantiated on UI thread, it is associated with *Looper's* message queue. On the other hand, messages posted to the message queue (1) holds a reference to the *Handler (3).* As a result, application framework can call *Handler i.e. handleMessage(Message)* whenever the message is processed by the *Looper*.

```
public class HandlerActivity extends AppCompatActivity {

    private final Handler mLeakyHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            Log.e( tag: "Abebaw",  msg: "handle message");
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.handler_activity_leak);

        // Post a message and delay its execution for 6 minutes
        mLeakyHandler.postDelayed(new Runnable() {
            @Override
            public void run() {
                Log.e( tag: "Abebaw",  msg: "in run()");
            }
        },  delayMillis: 1000 * 60 * 6);
    }
}
```

Figure 17: Inner Class Runnable Handler leaks

As it is shown in the code, after the activity is finished, the delayed message will continue in the UI thread's message queue (for six minutes) (2) before it is processed. The message (1) holds a reference to the activity's *Handler (3),* and the *Handler* holds an implicit reference to its outer class (*HandlerActivity*). This reference will exist until the message is processed. As a result, it prevents the activity context from being garbage collected and leads to leakage of the application resources.

5.3.6 Refactored Handler Activity Leak

If any handler in the program is alive, the bounding activity is also alive even though we might not need it, but we sacrifice the memory. Once the *Message* or *Runnable* is posted into *Handler*, then it is stored in *Message* referenced from *Looper Thread* till the message is executed. Posting a delay message is a leak for at least the value of the delay time. On the other hand, posting a message without a delay can cause a memory leak if the message queue is too large. Here, the lifetime of Handler and Activity is different.

The figure depicted below indicates the optimized version of Handler activity leak pattern. As numbering shows, (1) the message is posted and delayed for six minutes before execution, (2) in

order not to allow a thread to exist after execution the callback is removed in activity *onDestroy* method, (3) and (4) the runnable and Handler class is made static instead of inner class so that it doesn't have a reference to the containing activity. In addition to this, just like the previous solutions, using weak reference can also solve the problem in the same way.

```
public class HandlerActivity extends AppCompatActivity {

    private final Handler mLeakyHandler = new MyHandler();
    private final MyRunnable myRunnable = new MyRunnable();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.handler_activity_leak);

1       mLeakyHandler.postDelayed(myRunnable,  delayMillis: 1000 * 60 * 6);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();

2       mLeakyHandler.removeCallbacks(myRunnable);
    }

3   private static class MyRunnable implements Runnable {
        @Override
        public void run() { Log.e( tag: "Abebaw",  msg: "in run()"); }
    }

4   private static class MyHandler extends Handler {
        @Override
        public void handleMessage(Message msg) { Log.e( tag: "Abebaw",  msg: "handle message"); }
    }
}
```

Figure 18: Handler Activity Leak Optimization

5.4 Resource Leakage Through Static References

This leakage happens because of the reference of long living static objects. Objects that are tagged to the session may also have the same lifetime as the session, which is created per user and it may remain until the user logs out of the application.

5.4.1 Static Variable to an Instance of Inner Class Leak

An inner class is a case where a class definition is contained in another class. The dependency reference of the inner classes that may cause a memory leak is shown in Figure 19.

There are three types of inner classes (Inner class, Method Inner class, and Anonymous Inner class). In this study, we just analyzed the first category which is an inner class. The well-known properties of this inner class include;

- It may access the enclosing class members, such as private variables without the reference.
- Its lifetime is thought to be no longer than the enclosing (containing) class.
- Automatically it has an implicit reference to the outer class which will lead to memory leak.

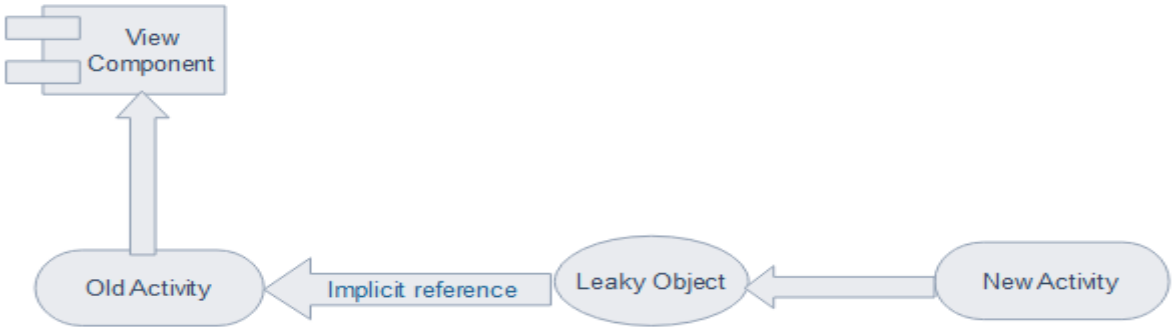


Figure 19: Dependency Reference graph of between objects in inner class leak pattern

To explain how the code in Figure 20 causes a leak:

- The leaky object (*someInnerClass*) (1) has an implicit reference to the container class (*StaticInnerClassActivity*).
- When clicking a button and return to the main menu, or when rotating a device, a configuration change will occur. In this scenario, the old activity will be destroyed, and a new activity will be recreated.
- The new activity (2) has a reference to a leaky object due to a static property of an object, and the leaky object has reference to old activity, GC will not collect old activity. This is because of the lifetime of the inner class object is become longer than its container.

```
public class StaticInnerClassActivity extends AppCompatActivity {

    1 private static SomeInnerClass someInnerClass;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.static_reference_to_inner_class_activity_leak);

        if (someInnerClass == null) {
            someInnerClass = new SomeInnerClass();
        }
    }

    2 class SomeInnerClass {
    }
}
```

Figure 20: Static Inner Class Leak Pattern Code Snippet

The code snippet indicates how the static variables to an instance of the inner class leak the memory through strong references.

5.4.2 Refactored Static Inner Class Leak

To solve a leaky property of an activity, the study considers making the static variable to non-static so that the new activity reference to the variable object is not strong and can be released from the memory when GC is called.

```
public class StaticInnerClassActivity extends AppCompatActivity {

    1 private SomeInnerClass someInnerClass;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.static_reference_to_inner_class_activity_leak);

        2 someInnerClass = new SomeInnerClass();
    }

    class SomeInnerClass {
    }
}
```

Figure 21: Static Variable to an Instance of Inner Class Leak Optimization

From the code, (1) removing the static keywords from member variable helps when there is a configuration change and new activity creation (i.e. (2)), the new activity references this private variable, and this variable doesn't have an implicit reference to the outer class so that whenever the GC is called this variable can be released from a memory.

5.4.3 Static View Activity Leak

Static view is all about maintaining a reference to a view. View leak is one of the worst android application leakages a programmer will face. View by itself is not a leak, but what they reference causes a leak.

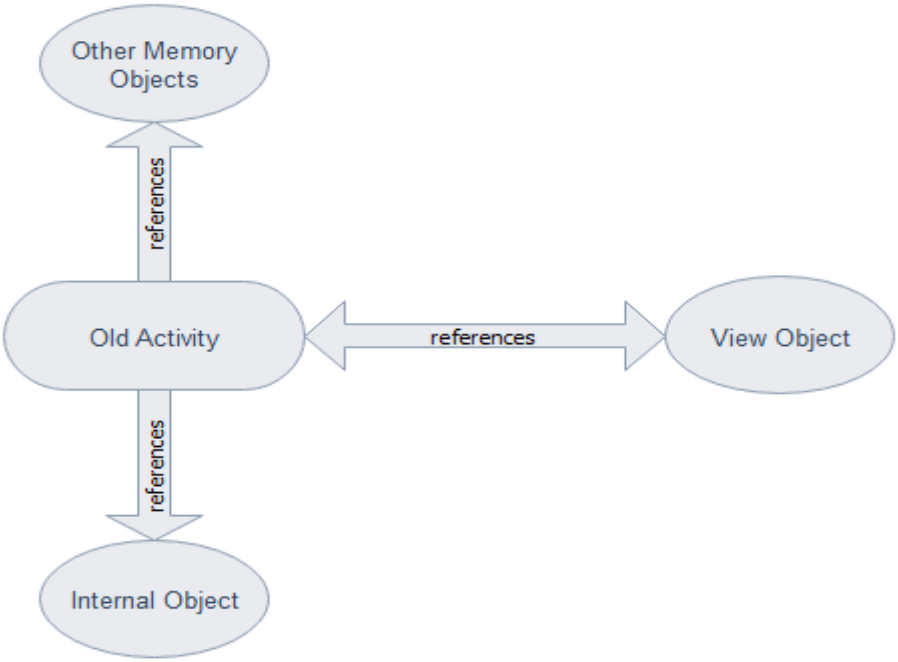


Figure 22: Dependency Reference for Static View Activity Leak

As indicated in the figure 22, view object contains a reference back to the activity that created it and the activity, in turn, references plenty of internal objects and other memory objects or items. Whenever the user clicks a *Static View Activity Leak* on UI depicted in Figure 7 or user rotates a device, configuration change will be triggered, which results in the activity to be destroyed, and the new instance of it will be loaded in the memory.

```
public class StaticViewActivity extends AppCompatActivity {


    static TextView label;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        label = new TextView( context: this);
        label.setText(getString(R.string.static_view_leak_explanation,
                getString(R.string.leak_check_instruction)));

        setContentView(label);
    }
}
```

Figure 23: Static View Activity Leak Pattern

But, the problem here is static view objects persist for the lifetime of an entire process that runs the app. The lifetime of this entire process is not the same as the lifetime of an activity that declares the static object. As a result, having a static object reference to a view object leads to leakage when the activity is destroyed.

5.4.4 Refactored Static View Activity Leak

A view object holds a reference to the containing activity. Having a static view object is equivalent to the static context object. Views by itself is not a leak problem rather than what they are referencing. View object contains a reference back to the activity that created it and the activity, in turn, references plenty of internal objects and other memory objects or items. As a result, if the leak occurs, it persists in the memory for runtime time of the application loaded in the memory. Static view is always available and never get killed. This study addresses this issue by making a view object to non-static so that it will be released from the memory.

```
public class StaticViewActivity extends AppCompatActivity {

    1 private TextView label;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        label = new TextView( context: this);
        label.setText(getString(R.string.leak_explanation_static_view));

        setContentView(label);
    }
}
```

Figure 24: Static View Activity Leak Optimization

From the code snippet, (1) the static view object is made non-static so that it will be eligible for garbage collection.

5.4.5 Static Variable Activity Leak

Static variables have the full lifetime of an activity that instantiates them. The objects then remain until the user terminates the application. Static variables are used as *roots* to the GC because they are referenced by their parent class. Hence, unless these variables are set to null, they will exist if program lives. So that everything is reachable from them. As a result, static variables cannot be elected for garbage collection while the class is loaded. They can be collected when the class that creates them is collected for the garbage.

```
public class StaticVariableActivity extends AppCompatActivity {

    1 static Activity activity = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.static_variable_activity_leak);
        TextView textView = (TextView) findViewById(R.id.textView);

        if (activity == null) {
            activity = this;
        }
    }
}
```

Figure 25: Static Variable Activity Leak Code Snippet

As indicated with the code snippet in Figure 25, the static variable activity is not eligible for garbage collection. Because the parent activity (*StaticVariableActivity*) references the static variable activity (1).

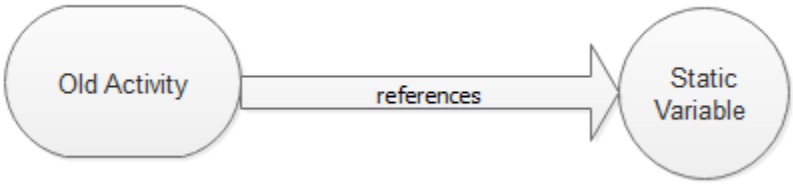


Figure 26: Dependency relation between a static variable and Old Activity

The old activity (parent activity that creates the static variable) references the static variable.

5.4.6 Refactored Static Variable Activity Leak

The simplest way of leaking an activity is by defining a static variable within a class definition of an activity and setting it to the running instances of that activity. To avoid the activity leakage, the reference should be cleared before the activities lifecycle completes. So, to make it eligible for the garbage collection, ensure if all its references are set to null. Because automatic garbage collection runs on the objects which are unused or the objects that are not referenced from any other running threads.

```
public class StaticVariableActivity extends AppCompatActivity {

    private Activity activity = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.static_variable_activity_leak_context);
        TextView textView = (TextView) findViewById(R.id.textView);

        if (activity == null) {
            activity = this;
        }
    }
}
```

Figure 27: Static Variable Activity Leak Optimization

As we see from the code, (1) removing static keyword makes the activity leak free. In addition to this, it is also possible to use a weak reference.

5.4.7 Singleton Object Activity Context Leak

Singleton pattern is a software design pattern that restricts the instantiation of a class to one object. It is essential when exactly one object is needed to coordinate activities throughout the system. To detect a memory leak due to singleton pattern, we have created two activities. *SingletonActivity* that have a button that opens *SingletonManager* activity alongside with a singleton class that holds a static reference of a *Context.*

```
public class SingletonActivity extends AppCompatActivity {

    SingletonManager singletonManager = null;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.singleton_activity_leak);

        singletonManager = SingletonManager.getInstance(this);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.e( tag: "Abebaw",  msg: "onDestroy: called");

    }
}
```

Figure 28: Singleton Activity code snippet that opens Singleton Manager

The singleton manager object that is going to be accessed from another class in the system (through *getInstance*) is defined in Figure 29.

```
public class SingletonManager {
    private static SingletonManager singletonManagerSingleton;
    private Context context;

    private SingletonManager(Context context) {
        this.context = context;
    }

    public static synchronized SingletonManager getInstance(Context context) {
        if (singletonManagerSingleton == null) {
            singletonManagerSingleton = new SingletonManager(context);
        }
        return singletonManagerSingleton;
    }

    public void unregister(Context context) {
        if (this.context == context) {
            this.context = null;
        }
    }
}
```

Figure 29: SingletonManager Activity

In the code, the context used is an activity context. So that when the user tries to rotate a device or press the button and click back after the first test, there are no leaks as the object is garbage collected. Because activity context is destroyed if the activity is destroyed. Here in the code, *getInstance ()* without *synchronized* is not thread-safe. So, to make it thread safe we included *synchronized* (to allow one thread at a time and to prevent concurrent access as well as corruption, modification of variables or data from various threads) keyword to the method.

When android context classes are placed in the static field that is the static reference to *SingletonManager* which has a field *context* pointing to *Context* as indicated in the program will lead to a leak.

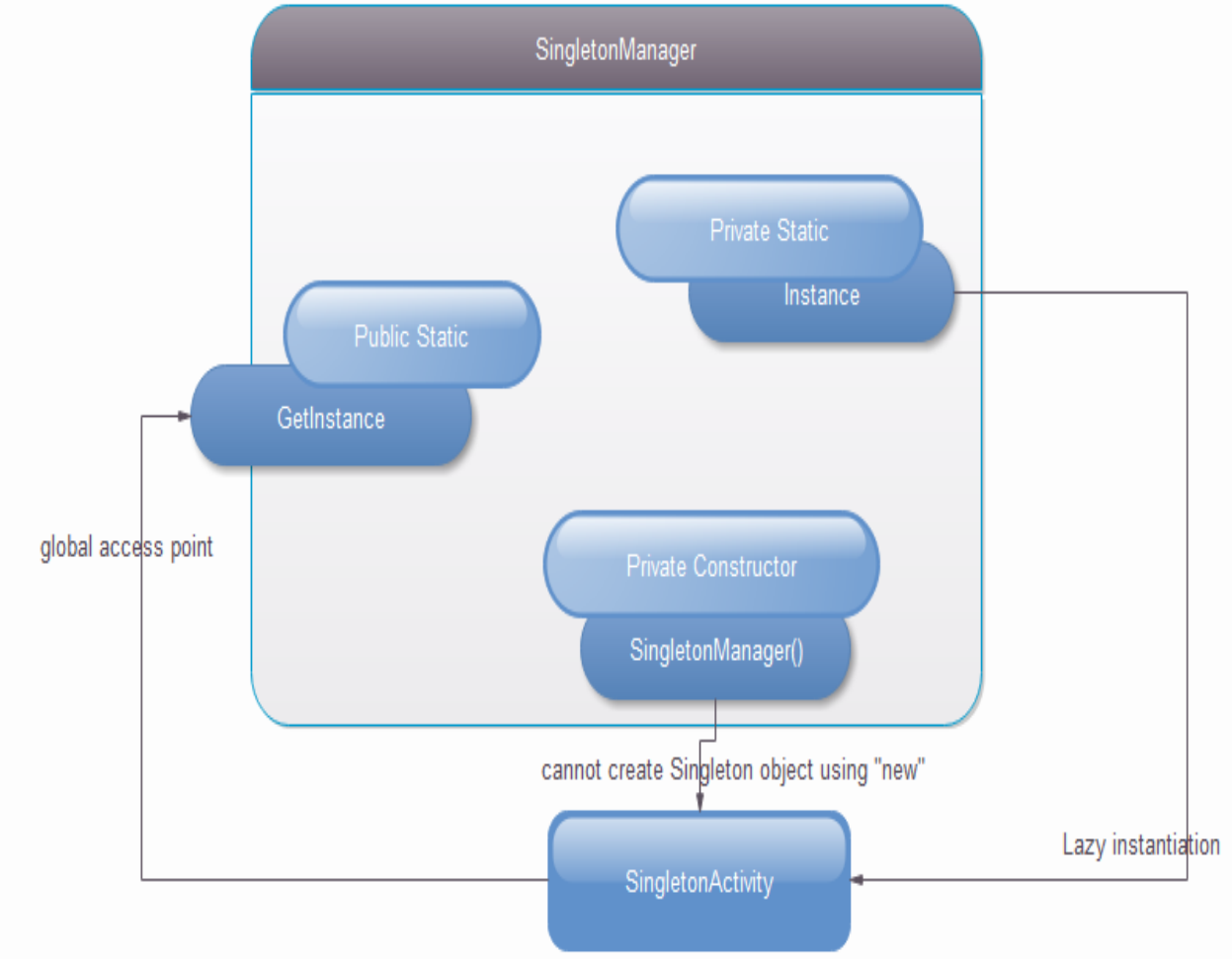


Figure 30: The nature of Created Singleton Pattern

As indicated in the figure 30, outside code cannot create a singleton object. In addition to this, instance variables ("private") so that it cannot be accessed outside the *SingletonManager"*.

Instance variables are “static” so object memory is created once and reused among all other objects only using *getInstance ()*. The other thing is that an instance in the program is created only when *getInstance ()* is called by the first request, or when needed (i.e. *Lazy Instantiation*).

5.4.8 Refactored Singleton Object Activity Context Leak

In a singleton pattern, the application context which is an instance of the singleton can be accessed through *getApplicationContext ()*. This application context is tied to the application lifecycle and this can be used when the context is needed whose lifecycle is isolated from the current context or when there is a need to pass a context beyond the scope of the activity. As a result, for every singleton object created for the application, since this object needs a context it is the best way to use an application context. Instead of passing application context, if the programmer passes the activity context, the reference to the activity will persist and it leads to the memory leak as it will not be garbage collected. If there is a context that lives longer, it is the best solution to use an application context.

Because the activity context is tied to the activity lifecycle. So, it should be used when there is a need to pass the context in the scope of the activity or when there is a need for the context whose lifecycle is attached to the current context. as a result, this case is not advisable for the singleton object since it is beyond the activity lifecycle. To address such issues, the codes are optimized in the following way.

```
public class SingletonActivity extends AppCompatActivity {

    SingletonManager someSingletonManager = null;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.singleton_activity_leak_context);
2       someSingletonManager = SingletonManager.getInstance(this);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.e( tag: "Abebaw",  msg: "onDestroy: called");
1       someSingletonManager.unregister( context: this);
    }
}
```

Figure 31: Optimized Singleton Object Activity Context Leak

In this code, (1) the singleton object is unregistered at onDestroy callback so that the activity object passed (2) is not leaked. Alternatively, in Figure 32 in addition to unregistering (1), instead of *this*,

getApplicationContext () is passed. As a result, the resource leakage is fixed. So, the programmer can use either of this depending on their interest.

```
public class SingletonActivity extends AppCompatActivity {

    SingletonManager someSingletonManager = null;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.singleton_activity_leak_context);
2       someSingletonManager = SingletonManager.getInstance(getApplicationContext());
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.e( tag: "Abebaw",  msg: "onDestroy: called");
1       someSingletonManager.unregister( context: this);
    }
}
```

Figure 32: Alternative Optimization for Singleton Object Activity Context Leak

5.5 Memory Performance Analysis

To enable the extraction of logcat data to the external directory and simplify the analysis of the memory consumption and effects of the memory leak patterns, the performance analysis script is created.

5.5.1 Performance Analysis Script

This is a monkeyrunner script used to evaluate the performance of application's using garbage collection calls and details of the collection process. The script provides a functionality of orchestrating the testing process and collecting several garbage collection calls with the description, automating the testing process, and analysis of the results found.

At the start of the testing process, the script defines the test parameter that should be considered to aid the testing, (1) extraction of test data and to write them to the external storage. In this parameter dictionary, (2) the analysis results that are not relevant to the current package is disabled to generate a precise output for the simplicity of the analysis

In Figure 33. before the execution of the test, the script instantiates the dictionary that will be used for evaluation of any packages of the application.

```
DUMPSYS_FILENAME = 'dumpsys.log'

def perform_test(device, package_name):
    print 'Starting test'

    params = dict({
    1   'listener': 'com.example.abebaw.performanceanalysis.TestListener',
    2   'disableAnalytics': 'true'
    })

    test_runner = (package_name + '.test' +
                   '/android.support.test.runner.AndroidJUnitRunner')
```

Figure 33: Execution parameters for the test

Then after in Figure 34, the permission of the device is declared in Android Manifest/Debug files. So, to enable dump permission on the debuggable test APK, the program is defined as follows in the script. It includes (1) Android device permission and (2) installed debuggable app permission for the extraction of test data. The storage permission follows the similar syntax, rather a keyword DUMP is replaced by WRITE_EXTERNAL_STORAGE. This permission should then be declared in the manifest file.

```
def enable_dump_permission(sdk_path, device_id, dest_dir, package_name):

    print 'Starting dump permission grant'
    perm_command = [os.path.join(sdk_path, 'platform-tools', 'adb'),
                    '-s', device_id,
                    'shell',
                    'pm', 'grant', package_name,
                1   'android.permission.DUMP']
2   log_file_path = os.path.join(dest_dir, 'logs', 'enable_dump_perm.log')
    with open(log_file_path, 'w') as log_file:
        try:
            subprocess.call(perm_command,
                            stdout=log_file,
                            stderr=subprocess.STDOUT,
                            shell=False)
        except OSError:
            print 'ERROR executing permission grant.'
```

Figure 34: A class to enable dump permission on debuggable test APK

To remove test files that have been previously executed it should be replaced with the current test data so that the test files need to be removed or cleaned.

```
def clean_test_files(dest_dir):
    print 'Cleaning data files'
    folders = [os.path.join(dest_dir, 'testdata'),
               os.path.join(dest_dir, 'logs')]
    for the_folder in folders:
        if os.path.isdir(the_folder):
            for the_file in os.listdir(the_folder):
                file_path = os.path.join(the_folder, the_file)
                try:
                    if os.path.isfile(file_path):
                        os.unlink(file_path)
                    elif os.path.isdir(file_path):
                        shutil.rmtree(file_path)
                except IOError, exception:
                    print exception
    for the_folder in folders:
        if not os.path.isdir(the_folder):
            try:
                os.makedirs(the_folder)
            except OSError:
                print 'ERROR Could not create directory structure for tests.'
```

Figure 35: Method to remove the previous test files

The apps for performance evaluation is then can be opened using the following code.

```
def open_app(device, package_name):

    device.shell('am start -n ' + package_name + '/' + package_name +
                 '.MainActivity')
```

Figure 36: Method to open the specified app on the device for evaluation

This class can also be called in every test execution in the main method using the following code

```
run_tests(sdk_path, device, device_id, dest_dir,
                    package_name)
```

Figure 37: Function call for executing test thread

To create and run the test for test thread that is defined using espresso and to collect memory information, the test thread is defined and executed.

```
def run_tests(sdk_path, device, device_id, dest_dir,
                                    package_name):
    """Create and start threads to run tests.
    """
    test_thread = threading.Thread(name='TestThread',
                                   target=perform_test,
                                   args=(device,
                                         package_name))
    test_thread.start()
    test_time_completion = int(time.time())
    print 'Test Thread Done'
```

Figure 38: Method to run test thread

5.6 Android Manifest/Debug Permission

To enable the files to be pulled to the external directory the permissions are defined in debug Android Manifest file as follows

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

<uses-permission android:name="android.permission.DUMP"
    tools:ignore="ProtectedPermissions" />
```

Figure 39: Permission declared for dumping and writing to external storage.

5.7 Instrumentation Test

The context of applications under test and Android components such as clicking button and application lifecycle which is intended to show the effects of each pattern on performance accessed by instrumentation for the analysis. This test is supported via JUnit runner and espresso test cases to access the instrumentation information.

5.7.1 Test Listener

The test listener enables log files to be retrieved by cleaning data from previous tests and to check the test is complete. This method is implemented in the following way:

```
@Override
public void testRunStarted(Description description) throws Exception {
    Log.w(LOG_TAG,  msg: "Test run started.");
    deleteExistingTestFilesInAppData();
    deleteExistingTestFilesInExternalData();


    super.testRunStarted(description);
}

@Override
public void testRunFinished(Result result) throws Exception {
    Log.w(LOG_TAG,  msg: "Test run finished.");

    super.testRunFinished(result);

    try {
         // Create a file indicating the test run is complete.
        File testRunFinishedFile = PerformanceTestingUtils.getTestRunFile( filename: "testRunComplete.log");
        testRunFinishedFile.createNewFile();
        Log.w(LOG_TAG,  msg: "testRunComplete file written.");

        copyTestFilesToExternalData();

        Log.w(LOG_TAG,  msg: "Done copying files to external data directory");
    } catch (Exception e) {
        Log.e(LOG_TAG,  msg: "Taking all log files after test run", e);
    }
}
```

Figure 40: Code to enable log files to be extracted

In addition to this, test listener enables to copy test files to an external directory with the help of performance analysis test script. Similarly, deletion of existing test files in external data and existing test files in app data performed on the test listener.

```
String srcAbsolutePath = PerformanceTestingUtils.getTestRunDir().getAbsolutePath();
String destAbsolutePath = externalTestFilesStorageDir.getAbsolutePath();

Log.w(LOG_TAG,  msg: "Moving test data from " + srcAbsolutePath + " to " + destAbsolutePath);

processBuilder.command("cp", "-r", srcAbsolutePath, destAbsolutePath);
processBuilder.redirectErrorStream();
Process process = processBuilder.start();
process.waitFor();
if (process.exitValue() != 0) {
    StringBuilder errOutput = new StringBuilder();
    char[] charBuffer = new char[1024];
    int readSize;
    InputStream errorStream = null;
    Reader reader = null;
    try {
        errorStream = process.getInputStream();
        reader = new InputStreamReader(errorStream);
        while ((readSize = reader.read()) > 0) {
            errOutput.append(charBuffer,  offset: 0, readSize);
        }
    } finally {
        if (errorStream != null) try { errorStream.close(); } catch (Exception ignored) {}
        if (reader != null) try { reader.close(); } catch (Exception ignored) {}
    }
    String errorString = errOutput.toString();
    Log.e(LOG_TAG, errorString);
    throw new IOException("Not able to move test data to external storage directory:"
            + " src=" + srcAbsolutePath + ", dest=" + destAbsolutePath + ", out=" +
            errorString);
}
```

Figure 41: Copy test files to external data

5.7.2 Performance Testing Utility

It is a common and centralized way which helps to ensure every test perform an action in the same manner. To retrieve the app under test and write the test files for each test cases specified with espresso associated with their class name and test cases this class is implemented for the sake of analysis simplicity for each test.

```
    public static Context getAppContext() {
        return InstrumentationRegistry.getInstrumentation().getTargetContext();
    }
    public static File getTestRunFile(String filename) {
        return new File(getTestDir( className: null,  testName: null), filename);
    }

     public static File getTestRunDir() {
        return getTestDir( className: null,  testName: null);
    }
    private static String getTranslatedTestName(String className, String testName) {
        if (className == null || testName == null) {
            return null;
        }
        String base = className + "_" + testName;

        base = base.replace( target: "com",  replacement: "c")
                .replace( target: "example",  replacement: "e")
                .replace( target: "abebaw",  replacement: "a")
                .replace( target: "performanceanalysis",  replacement: "pa");

        return base;
    }

    public static File getTestFile(String className, String testName, String filename) {
        return new File(getTestDir(className, testName), filename);
    }
}
```

Figure 42: A Centralized method for which every test classes perform

The output of this method for every test classes is written to the external directory in the following way. This test data contains the detail information about the memory consumption, reference and number of garbage collection call with associated pause time.

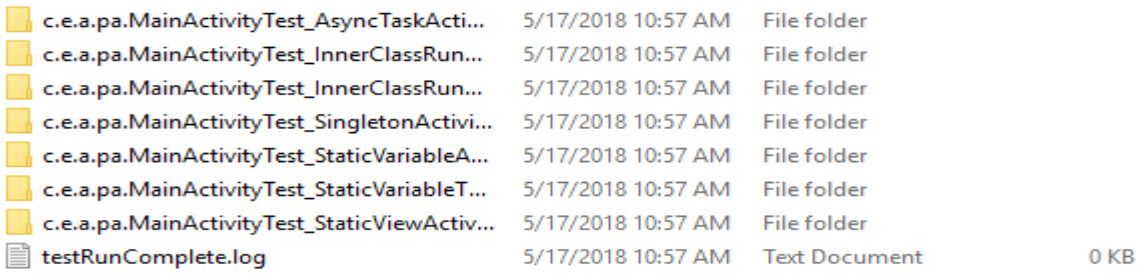


Figure 43: Test files on the external directory

5.7.3 Performance Test Annotation

Wherever the annotation is defined within a test case, the performance tests defined within a project will be performed. Annotation of this performance analysis program is specified as *PerformanceTest* to annotate a test that contains a performance component as follows:

```
public @interface PerformanceTest {
}
```

Figure 44: Performance test annotation for the program

5.7.4 Espresso Test Cases

To evaluate the resource leak patterns for their memory leakage performance issues and to instantiated automatically using local performance tests test cases look like in Figure 45 is specified for each class including performance test annotation and JUnit Test annotation (which enables to perform Instrumentation testing). The code inside colored rounded rectangle calls the command that enables to dump logcat information from the device. Whereas, (1) is a timing parameter that waits two minutes until the dumping process finishes. This is not the exact time because dumping process depends on the number of activities available for execution.

```
@RunWith(AndroidJUnit4.class)
 @LargeTest

 @PerformanceTest
public class MainActivityTest {

    @Rule
    public EnableLogcatDump mEnableLogcatDump = new EnableLogcatDump();

    @Test
    @PerformanceTest
    public void StaticViewActivityTest() throws InterruptedException {
        onView(withId(R.id.buttonLeakActivityStaticView)).perform(click());
        pressBack();
        Thread.sleep( millis: 2*60*1000);

    }
```

Figure 45: Test cases for Static View Activity leak pattern

During the evaluation, when the button automatically clicked, or device is rotated by the user, the view should have to be displayed to show clearly the impacts of those patterns in the activity lifecycle. In case of device rotation, the device should be rotated after the button is clicked and associated view is displayed.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);


    Button buttonLeakActivityStaticView =  findViewById(R.id.buttonLeakActivityStaticView);
    buttonLeakActivityStaticView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Intent intent = new Intent(getApplicationContext(), StaticViewActivity.class);
            startActivity(intent);
        }
    });
```

Figure 46: The view that will be displayed when performing an activity

5.7.5 Test Rules

The rules that are called while espresso test case starts in Figure 45 enables to execute tests and records the output of the test results at the end of the test for every test cases. A test rule for extracting logcat information that consists activity lifecycle and garbage collection calls is depicted in the code below.

```
 // Clear logcat buffer prior to test run.
public void before() throws Throwable {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        ProcessBuilder processBuilder = new ProcessBuilder();
        processBuilder.command("logcat", "-c");
        processBuilder.redirectErrorStream();
        Process process = processBuilder.start();
        process.waitFor();
        if (process.exitValue() != 0) {
            Log.e(LOG_TAG,  msg: "Error while clearing logcat, exitValue=" + process.exitValue());
        }
    }
}
// Extract logcat buffer to a file after test run.
public void after() {
    try {
        if (android.os.Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR2) {
            Trace.beginSection( sectionName: "Taking logcat");
        }
        ProcessBuilder processBuilder = new ProcessBuilder();

        processBuilder.command("logcat", "-d",
                "-f", PerformanceTestingUtils.getTestFile(mTestClass, mTestName,  filename: "logcat.log")
                        .getAbsolutePath());
        processBuilder.redirectErrorStream();
        Process process = processBuilder.start();
        process.waitFor();
        if (process.exitValue() != 0) {
            Log.e(LOG_TAG,  msg: "Error exit value while extracting logcat, exitValue=" +
                    process.exitValue());
        }
```

Figure 47: Rules for dumping logcat information

5.8 Custom Gradle Plugin

This plugin automates the creation of performance tests, by checking for changes whenever there is an alteration or modification of the code in the continuous delivery. This test automation mechanism includes two components: local performance test execution and performance test task generator plugin.

5.8.1 Execute Local Performance Tests

To execute local performance tests on the specific device that a test program is installed, first, a device Id must be pulled rather than setting it manually.

```
public class ExecuteLocalPerformanceTestsTask extends DefaultTask {
    Logger mLogger = getLogger()


    String mDeviceId = ""

    public ExecuteLocalPerformanceTestsTask() {
        super()
        setGroup('verification')
        setDescription("Run performance tests on a specific device.")
        if (deviceId != null) {
            mDeviceId = deviceId
        }
        getOutputs().upToDateWhen( closure: { return false })
    }
```

Figure 48: A method of setting a connected device Id automatically for testing

After setting a device id, the local performance test is executed by locating the path of the monkeyrunner in SDK's directory and locating the performance analysis test script which is executed or triggered by the monkeyrunner. The task of the local performance test is specified in the following way in Figure 49.

```
    def sdkDir = properties.getProperty('sdk.dir')

    def monkeyExt = ''
    if (Os.isFamily(Os.FAMILY_WINDOWS)) {
        monkeyExt = '.bat'
    }

    def monkeyPath = Paths.get(sdkDir,  ...strings: "tools\\bin", "mymonkeyrunner" + monkeyExt).toAbsolutePath().toString()
    def rootDir = getProject().getRootDir().getAbsolutePath()
    def monkeyScriptPath = Paths.get(rootDir,  ...strings: "performance_analysis.py").toAbsolutePath().toString()
    processBuilder.command(monkeyPath, monkeyScriptPath, rootDir, mDeviceId)
    processBuilder.environment().put( k: "ANDROID_HOME", sdkDir)
    processBuilder.redirectErrorStream()
    Process process = processBuilder.start()
    process.waitFor()

    // Redirect output from the script to the console.
    process.in.eachLine() { line ->
        mLogger.warn("Script: " + line)
    }
    if (process.exitValue() != 0) {
        throw new GradleException("Monkeyrunner script didn't complete")
    }
    mLogger.warn("Monkeyrunner complete")
}

public void setDeviceId(String deviceId) {
    setDescription("Run performance tests on device with serial ${deviceId}")
    mDeviceId = deviceId
}
```

Figure 49: Execution of performance analysis script using a monkeyrunner

5.8.2 Performance Test Task Generator Plugin

This plugin helps to define the activity that should run after completing performance test tasks, creates parent performance test task that can execute all the device specific tasks, creates performance test tasks for each connected device to the system, verify each device-dependent (specific) tasks run by parent performance test task. This is synced using “apply” in the build Gradle file. To get a connected device list for custom Gradle:

```
private List<String> getConnectedDeviceList(Project project) {
    def rootDir = project.rootDir
    def localProperties = new File(rootDir,  s: "local.properties")
    def sdkDir = ""
    if (localProperties.exists()) {
        Properties properties = new Properties()
        localProperties.withInputStream { instr ->
            properties.load(instr)
        }
        sdkDir = properties.getProperty('sdk.dir')
    }
    String adbCommand = sdkDir + File.separator + "platform-tools" + File.separator + "adb"
    logger.info("Using ADB command: ${adbCommand}")
    // Compose a list of connected Android devices.
    ArrayList<String> devices = new ArrayList<String>()
    ProcessBuilder processBuilder = new ProcessBuilder()
    processBuilder.command( ...strings: adbCommand, "devices", "-l")
    Process process = processBuilder.start()
    process.waitFor();
```

Figure 50: A method to get connected device list

This test mechanism is synced with the program in Gradle build file in the following way:

```
import com.example.abebaw.performanceanalysis.PerformanceTestTaskGeneratorPlugin

apply plugin: 'com.android.application'
apply plugin: PerformanceTestTaskGeneratorPlugin
```

Figure 51: Integration of Custom Gradle plugin with the project in build Gradle file

The local performance test is created in this class. The test dependent tasks, posttest tasks (to uninstall this dependent task after performance test), Creation of a parent performance test task that can run all the device-specific tasks, creation of a performance test task which can run on all connected devices and ensuring device-specific tasks are executed by parent performance test task are defined in the following code snippet.

```
private void createLocalPerformanceTestTasks(Project project) {
 ArrayList<Task> createdTasks = new ArrayList<Task>()
 List<String> connectedDevices = getConnectedDeviceList(project)
 HashSet<String> dependentTasks = new HashSet<String>();
 dependentTasks.add("assembleDebug")
 dependentTasks.add("assembleAndroidTest")
 dependentTasks.add("installDebug")
 dependentTasks.add("installDebugAndroidTest")
 logger.info("Dependent tasks found: " + dependentTasks)

 HashSet<String> postTestTasks = new HashSet<String>();

 postTestTasks.add("uninstallAll")
 logger.info("Post test dependent tasks found: " + postTestTasks)

 Task executeLocalPerformanceTests = project.tasks.create(name: 'executeLocalPerformanceTests',
         type: DefaultTask,
         group: 'verification',
         dependsOn: dependentTasks,
         description: 'Execute performance tests on all connected devices.')

 connectedDevices.each { androidDeviceId ->
     ExecuteLocalPerformanceTestsTask newTask = (ExecuteLocalPerformanceTestsTask) project.tasks.create(
             name: ('executeLocalPerformanceTests_' + androidDeviceId),
             type: ExecuteLocalPerformanceTestsTask)
     newTask.deviceId = androidDeviceId

     executeLocalPerformanceTests.dependsOn(newTask)
```

Dependent tasks before startup

Tasks to be displayed on Gradle

Figure 52: Local Performance test task creation for connected devices

After applying the code stated in Figure 51 and syncing with the project, a local performance test is ready in the Gradle projects as follows.

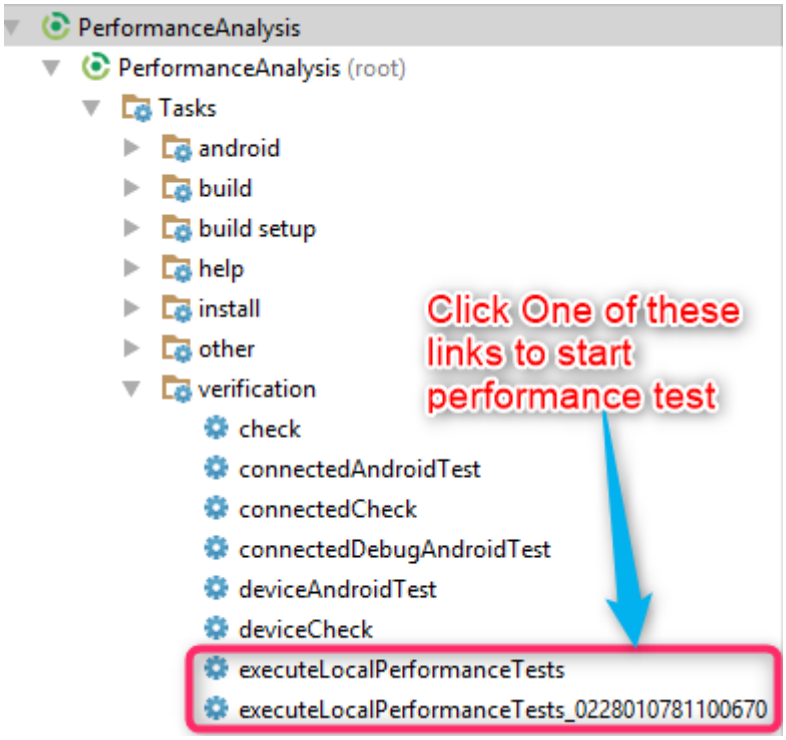


Figure 53: Gradle test for executing test on the connected device automatically

As indicated in the figure, when the user clicks *executeLocalPerformanceTests*, the test will be started automatically. Then, after installing dependent tasks and performing beforementioned tasks, this test will call the monkeyrunner script (i.e. performance analysis script).

CHAPTER SIX

EVALUATION, RESULT, AND DISCUSSION

In this chapter, the proposed solution for identified resource leak patterns impact on memory consumption in chapter five is empirically evaluated. In addition to this, the research questions raised in chapter one is discussed against the result of the study.

6.1 Resource Leakage Pattern in Long-Running Activities

Long-running threads (worker threads) live longer than an activity. If the reference of an activity takes place directly or indirectly from the worker thread, which outlives the activity, then the activity object leakage is in place.

When the android application has a serious memory leakage, a simple and short GC will not reclaim the resources, and the heap keeps on increasing that will force a larger GC to start. The larger GC pauses the entire application UI thread for around 100 ms to 200 ms. In this situation, the app will seriously lag and will become unusable. If the situation is not fixed, then the heap memory will constantly increase to the point where no memory is available for allocation of running app or OOM that crashes an app. In this study, the pause time for GC is minimal because it is evaluated on the impacts of specific leak pattern on memory. As the interaction of threads and activities within an app increases and awaits long, then the leakage and the pause time will increase too.

```
05-08 10:44:27.174 12377-12377/com.example.abebaw.performanceanalysis
 D/LifecycleMonitor: callback completes: android.support.test.rule
 .ActivityTestRule$LifecycleCallback@a1d32d5
05-08 10:44:32.243 12377-12394/com.example.abebaw.performanceanalysis I/art:
 Starting a blocking GC Explicit
05-08 10:44:32.255 12377-12394/com.example.abebaw.performanceanalysis I/art:
 Explicit concurrent mark sweep GC freed 17211(1196KB) AllocSpace objects,
 0(0B) LOS objects, 34% free, 7MB/11MB, paused 719us total 12.397ms
```

Figure 54: Call for GC and Pause time

6.1.1 AsyncTask Activity Leak

Having specified a global timeout period of testing to 20 seconds, since AsyncTask is a long running activity, it fails as shown in Figure 55.

```
AsyncTaskActivityTest(com.example.abebaw.performanceanalysis.MainActivityTest)
org.junit.runners.model.TestTimedOutException: test timed out after 20 seconds
    at java.lang.Object.wait(Native Method)
    at java.lang.Thread.parkFor$(Thread.java:2135)
    at sun.misc.Unsafe.park(Unsafe.java:358)
    at java.util.concurrent.locks.LockSupport.park(LockSupport.java:190)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2059)
    at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:442)
    at android.support.test.espresso.InteractionResultsHandler.gatherAnyResult(InteractionResultsHandler.java:83)
    at android.support.test.espresso.InteractionResultsHandler.gatherAnyResult(InteractionResultsHandler.java:52)
    at android.support.test.espresso.ViewInteraction.waitForAndHandleInteractionResults(ViewInteraction.java:310)
    at android.support.test.espresso.ViewInteraction.desugaredPerform(ViewInteraction.java:173)
    at android.support.test.espresso.ViewInteraction.perform(ViewInteraction.java:114)
    at com.example.abebaw.performanceanalysis.MainActivityTest.AsyncTaskActivityTest(MainActivityTest.java:93)
    at java.lang.reflect.Method.invoke(Native Method)
    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)
    at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
    at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)
    at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
    at org.junit.internal.runners.statements.FailOnTimeout$CallableStatement.call(FailOnTimeout.java:298)
    at org.junit.internal.runners.statements.FailOnTimeout$CallableStatement.call(FailOnTimeout.java:292)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.lang.Thread.run(Thread.java:764)
```

Figure 55: Test Failure Log file for AsyncTask Activity

The test log failure indicated in the figure has happened due to the longevity of the AsyncTask activity. When the global timeout period which is a time frame that the overall test cases should finish execution is passed (i.e. 20 seconds), then the activities that will not finish its execution like the AsyncTask activity will display the log error files to notify as it will not finish its operation within a specified time frame. This situation usually happens for the activities that last longer.

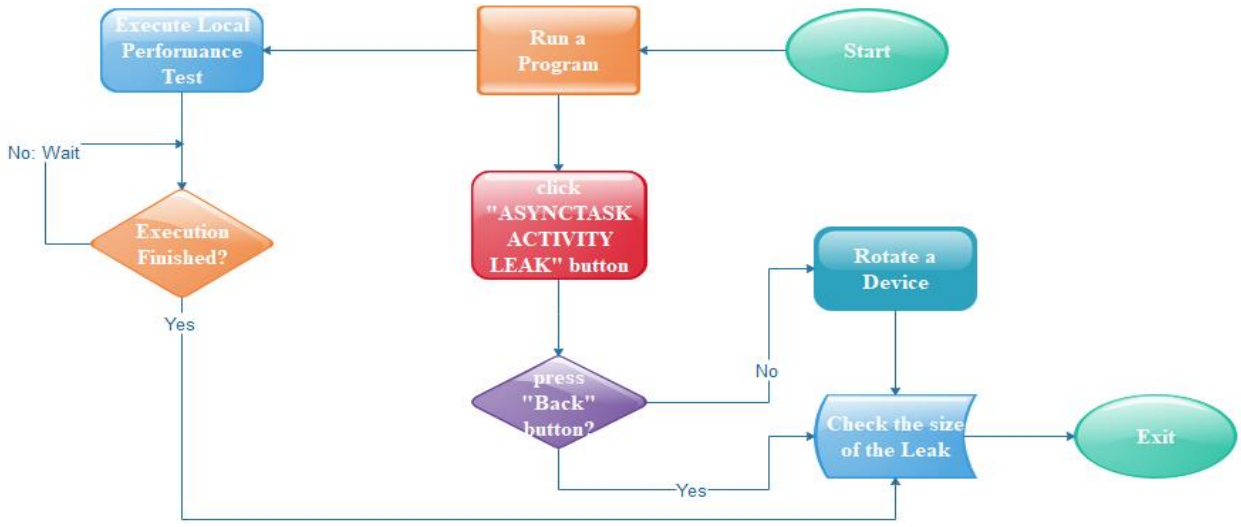


Figure 56: Leak Identification and Testing Flowchart for AsyncTask activity leak

This flowchart is applied for all identified and developed memory leak patterns except the button UI to be clicked that is associated with the corresponding leak patterns. Using this procedure, the leak pattern is tested, and the results are illustrated in Table 5. Each of the five tests are the tests which are conducted on activities within an app.

Table 6: AsyncTask Leak in KB in each test

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	8.2	8.2	8.2	8.2	8.2
Nexus 6 API 19	4.4	4.5	4.5	4.5	4.5	4.5
TECNO Camon CX API 24	7.0	57	57	57	57	57
Nexus 6P API 23	6.0	9.1	9.1	9.1	9.1	9.1
Nexus 5X API 17	4.2	4.6	4.6	4.6	4.6	4.6

The test result stated under Table 5 is found when the activity of AsyncTask class is destroyed after it is instantiated by user actions that are clicking on AsyncTask activity leak and returning to the main activity UI and device rotation.

6.1.2 Refactored AsyncTask Activity Leak

The evaluation mechanism follows the same process stated in the flowchart shown in Figure 56. So, we need to check the variation of the test results observed from five consecutive tests of the optimized version of codes.

Table 7: Optimized AsyncTask Activity Test Result in (KB) if any

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	0	0	0	0	0
Nexus 6 API 19	4.4	0	0	0	0	0
TECNO Camon CX API 24	7.0	0	0	0	0	0
Nexus 6P API 23	6.0	0	0	0	0	0
Nexus 5X API 17	4.2	0	0	0	0	0

The test result shows that whenever the garbage collector is called, the AsyncTask activity doesn't reference the enclosing activity object as the previous AsyncTask activity code pattern. As the

result, the leak is removed. As suggested in Chapter Five, the test result found in an alternative weak reference solution also the same with the first one. As a result, there are no variations in each of the five consecutive tests recorded in the testing process.

6.1.3 Thread Activity Leak

The general procedure of executing automated local performance tests on thread activity leak and identification of the causes of the resources leaks within a program as well as the identification of the amount of memory consumed due to thread leak activity is depicted with the flowchart in Figure 57.

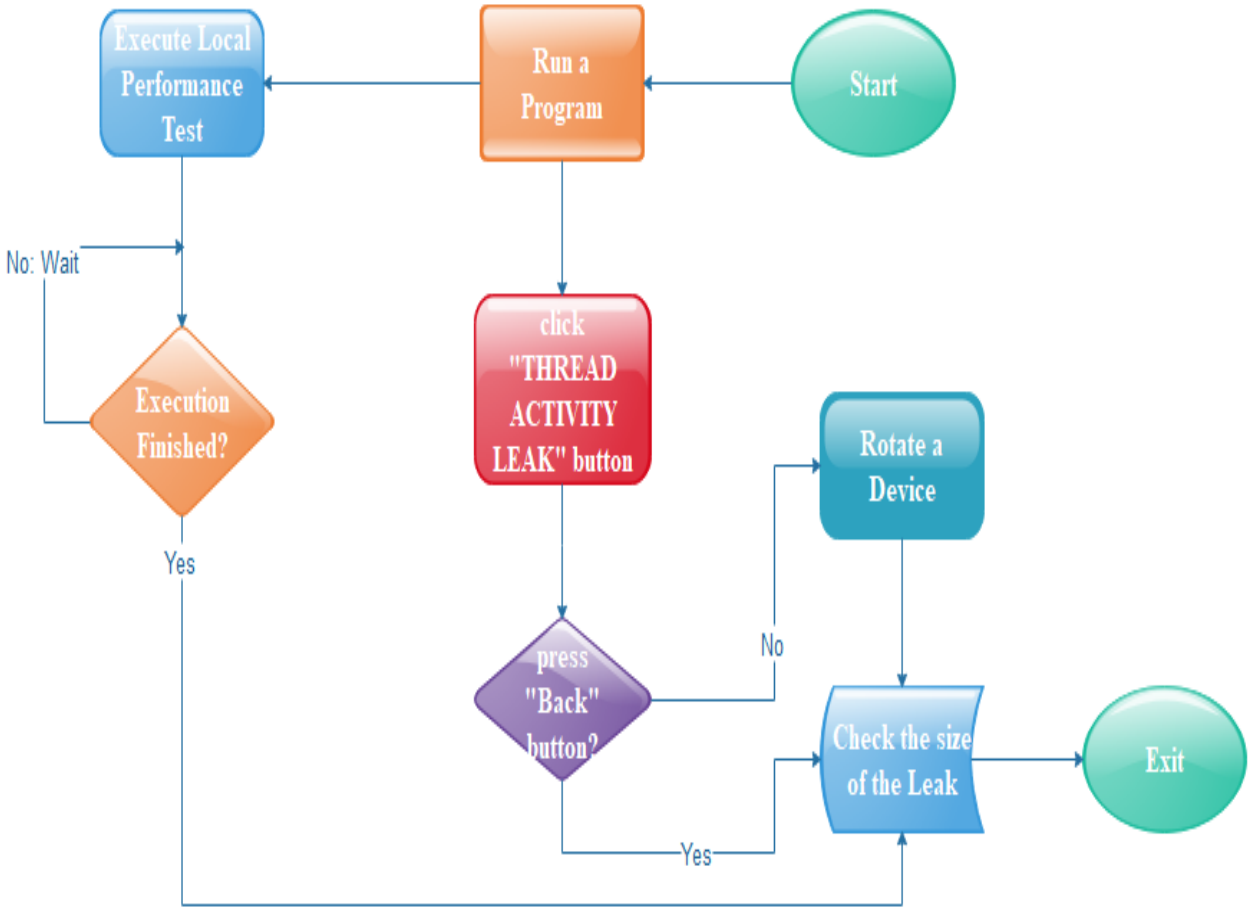


Figure 57: Leak Identification and Testing flowchart for Thread activity leak

The test results for the consecutive five times with different android versions and APIs are shown in Table 6. As it is indicated with the table, the leak is not stopped after the activity is destroyed. It continues throughout the application lifetime.

Table 8: Runnable thread activity leak results in (KB)

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	54	54	54	54	54
Nexus 6 API 19	4.4	110	80	76	92	81
TECNO Camon CX API 24	7.0	57	57	57	57	57
Nexus 6P API 23	6.0	81	81	81	53	53
Nexus 5X API 17	4.2	46	73	37	56	56

6.1.4 Refactored Thread Activity Leak

The procedure of testing is the same with the processes depicted in Figure 57. Tests are performed on five devices and emulators with different target android for the five consecutive times. The test results found from five consecutive tests on different devices and emulators with a different target Android is written in the table below.

Table 9: Optimized Thread Activity Test Result in (KB)

Device name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	0	0	0	0	0
Nexus 6 API 19	4.4	0	0	0	0	0
TECNO Camon CX API 24	7.0	0	0	0	0	0
Nexus 6P API 23	6.0	0	0	0	0	0
Nexus 5X API 17	4.2	0	0	0	0	0

The result indicates that with a simple modification on source codes can improve huge memory consumption of certain threads.

6.1.5 Handler Activity Leak

Just like the previous analysis of leak patterns, this memory leak pattern follows the same procedures except for the button UI that is going to be clicked by espresso test cases while starting the local automated performance tests on the connected devices.

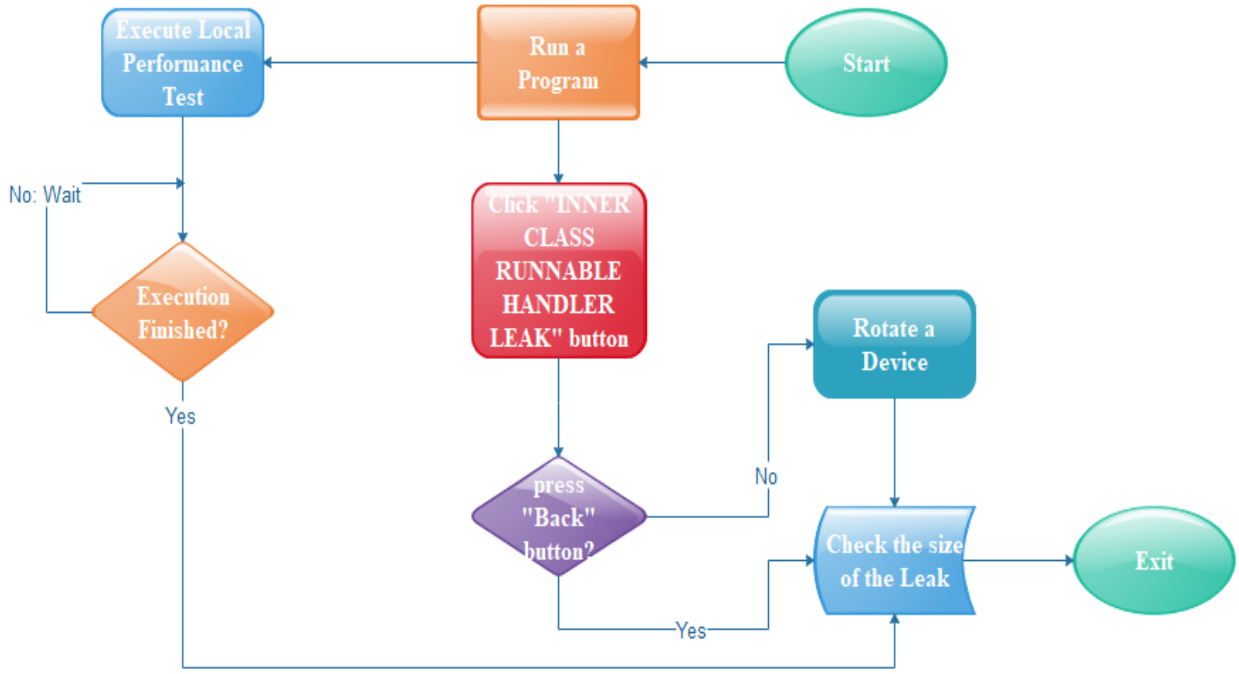


Figure 58: An Inner class runnable handler leak pattern detection and testing flowchart

The amount of memory leaked in kilobyte while executing simple *Handler Activity* is presented with emulators and devices under a different version of API and Android in Table 9.

Table 10: Inner Class Runnable Handler Leaks in (KB)

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	54	54	54	54	54
Nexus 6 API 19	4.4	76	63	63	63	63
TECNO Camon CX API 24	7.0	57	57	57	57	57
Nexus 6P API 23	6.0	81	9.1	9.1	9.1	9.1
Nexus 5X API 17	4.2	81	54	54	54	54

6.1.6 Refactored Handler Activity Leak

The testing mechanism is the same with Figure 58. As a result, the test is performed on five consecutive tests with different target Android and devices, the following results have been recorded.

Table 11: Optimized Handler Activity Test Result in (KB)

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	0	0	0	0	0
Nexus 6 API 19	4.4	0	0	0	0	0
TECNO Camon CX API 24	7.0	0	0	0	0	0
Nexus 6P API 23	6.0	0	0	0	0	0
Nexus 5X API 17	4.2	0	0	0	0	0

The test results indicated in the table also shows that refactoring of handler activity leaks with recommended performance tips can eliminate the possible huge amounts of leaks.

6.2 Resource Leakage Pattern in Static References

The static reference lives the lifetime of android application in a memory. Activity lifecycles are usually recreated and destroyed multiple times throughout the application's lifecycle. If the reference is taking place directly or indirectly from the static reference, there is no possibility of garbage collection after it has been destroyed. Activity size ranges from kilobytes to huge megabytes depending on the contents within it.

6.2.1 Static Variable to an Instance of Inner Class Leak

In the same way to the long-running activities, leakage due to a static variable to an instance of inner class leak pattern automated test for the detection of the amount of memory consumption identification is performed with the procedures depicted in the flowchart below.

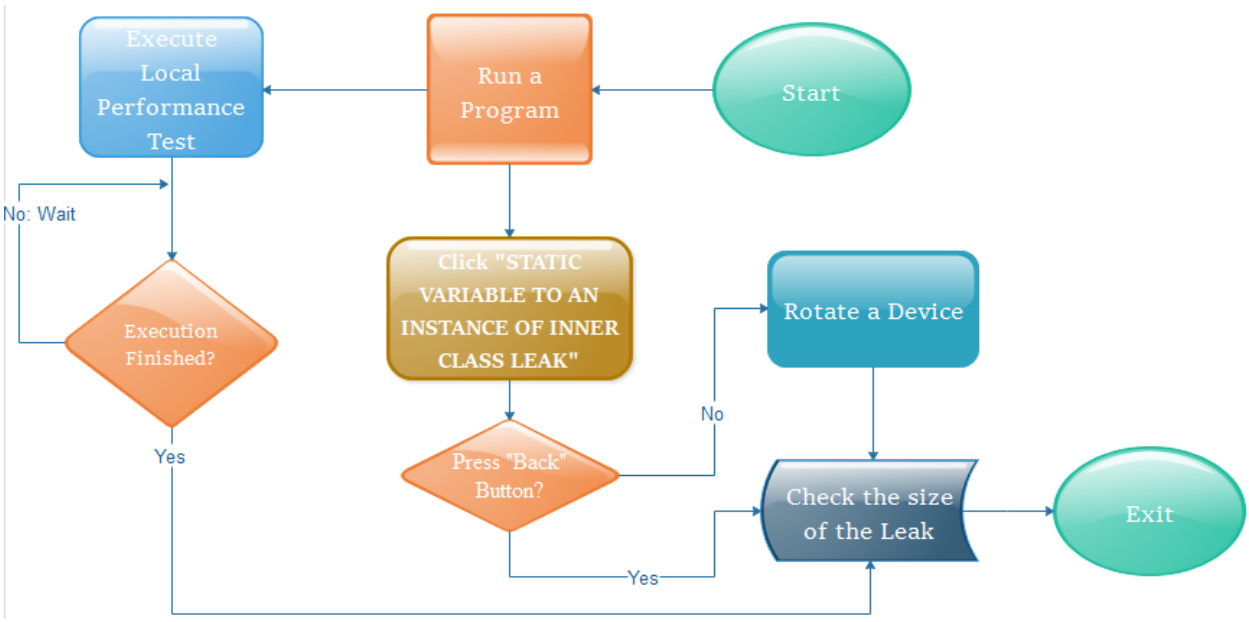


Figure 59: Static Variable to an Instance of Inner class Pattern Leak detection flowchart

The flowchart is the general procedures to indicate the size of the memory leak triggered by static variables to an instance of the inner class. Using the procedures in the flowchart in Figure 58, the result of the leakage in kilobyte is written under Table 11.

Table 12: Static Variable to an Instance of Inner Class Leak in (KB)

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	54	0	0	0	0
Nexus 6 API 19	4.4	45	0	0	0	0
TECNO Camon CX API 24	7.0	57	0	0	0	0
Nexus 6P API 23	6.0	37	0	0	0	0
Nexus 5X API 17	4.2	46	0	0	0	0

As it is seen in Table 11, the leakage after the first rotation or clicking of a button is zero. This is because after the first garbage collection call with a rotation the leaky object will be destroyed so that there will not be leakage.

6.2.2 Refactored Static Variable to an Instance of Inner Class Leak

Apart from section 6.2.1, the test is performed after the code optimization of the leak pattern. The results found in the test is illustrated in the following table.

Table 13: Optimized Static Variable to an Instance of Inner Class Test Result in (KB)

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	0	0	0	0	0
Nexus 6 API 19	4.4	0	0	0	0	0
TECNO Camon CX API 24	7.0	0	0	0	0	0
Nexus 6P API 23	6.0	0	0	0	0	0
Nexus 5X API 17	4.2	0	0	0	0	0

From the test results in the table, it is possible to conclude that the applied means of optimization on inner classes are effective in reducing the memory consumption of leaky patterns.

6.2.3 Static View Activity Leak

The reason why static view objects persist its leaking behavior after the first test in Table 13 unlike static variable to an instance of the inner class in Table 12 and static variable leak in Table 15 is that the view object persists to the applications lifetime.

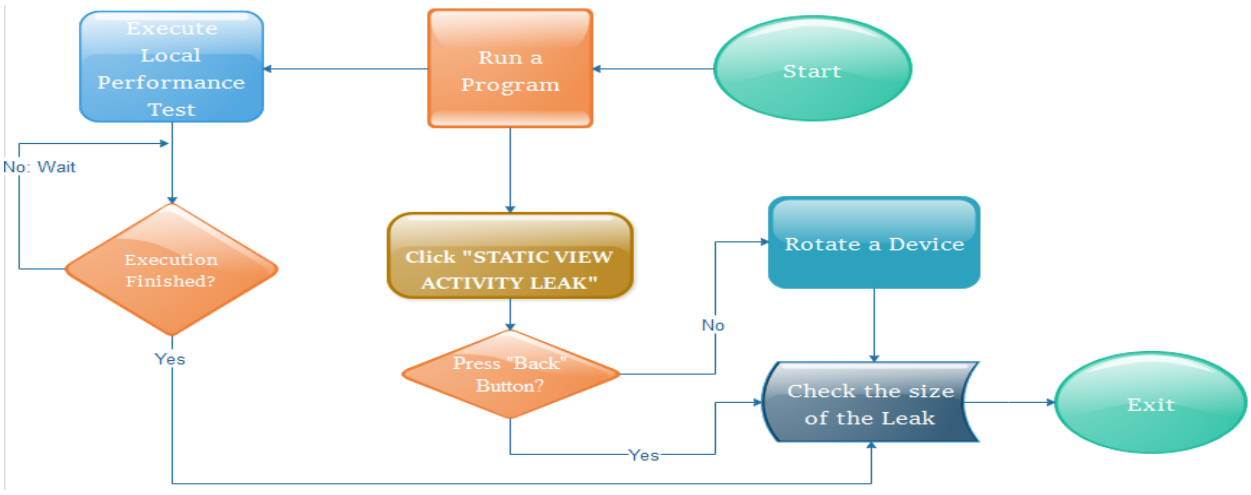


Figure 60: Leak identification and testing flowchart for Static View Leak Pattern

Using the procedures indicated in the flowchart, tested memory leaks in kilobyte while using the programming approaches mentioned in chapter five are illustrated in Table 13 below.

Table 14: Static View Activity Leak test results in (KB)

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	8.2	8.2	8.2	8.2	8.2
Nexus 6 API 19	4.4	32	4.5	32	4.5	4.5
TECNO Camon CX API 24	7.0	8.2	8.2	8.2	8.2	8.2
Nexus 6P API 23	6.0	9.1	9.1	9.1	37	9.1
Nexus 5X API 17	4.2	4.6	4.6	4.6	4.6	20

The test results show that leak due to static view is like leaking due to long-running activities as it lasts to the entire application lifetime.

6.2.4 Refactored Static View Activity Leak

After applying the optimization mechanisms in Chapter five for static view activity leak pattern, the memory leak is removed from the developed app in the following way.

Table 15: Optimized Static View Activity Leak Test Result in (KB)

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	0	0	0	0	0
Nexus 6 API 19	4.4	0	0	0	0	0
TECNO Camon CX API 24	7.0	0	0	0	0	0
Nexus 6P API 23	6.0	0	0	0	0	0
Nexus 5X API 17	4.2	0	0	0	0	0

6.2.5 Static Variable Activity Leak

The amount of memory consumed due to the leakage of the static variable object in the activity is tested in the following way.

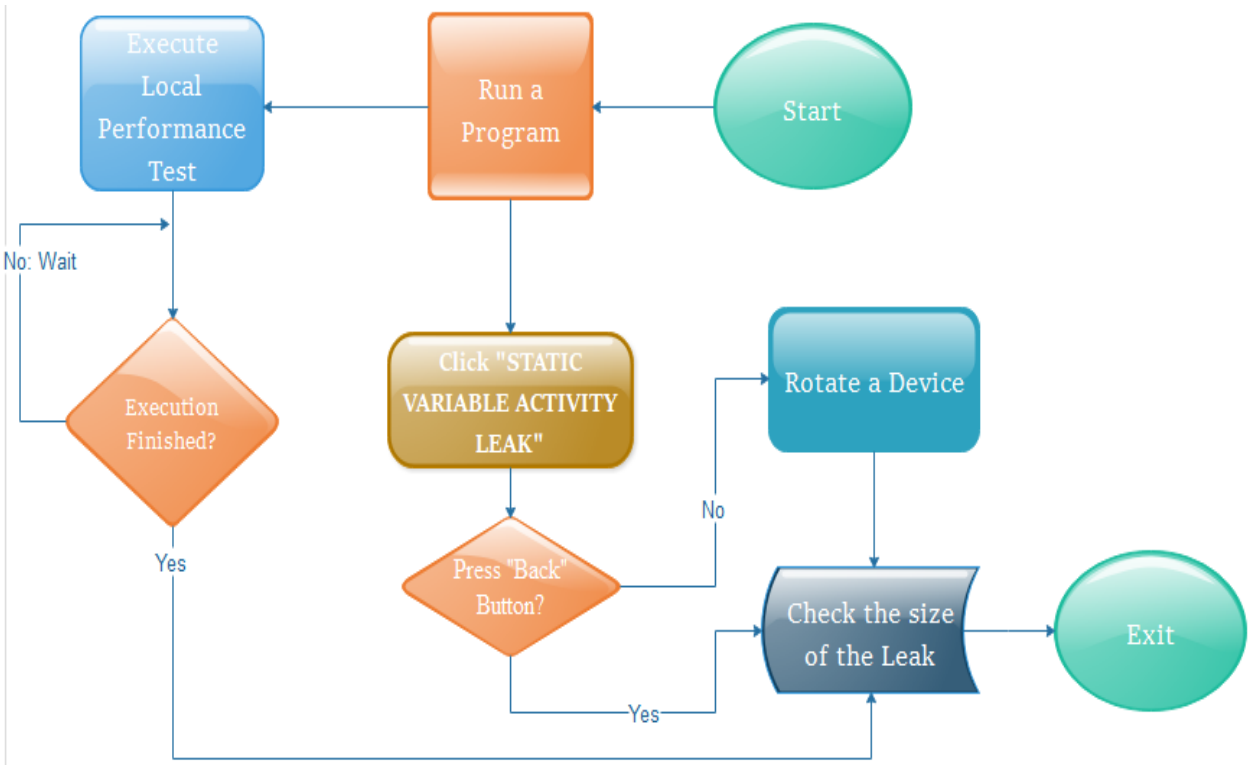


Figure 61: Leak Identification and Testing flowchart for Static Variable Leak Pattern

After the first test, there is no leakage. Because the activity that references the static variable is garbage collected, the variable remains unreferenced. So that, whenever there is a device rotation or button click and press back to the main menu, these variables are easily collected as a garbage. The test result of Test1 is different from the rest of the tests. Because, in static variable activity, the memory is allocated only during compilation time. As a result, there is configuration changes plenty of times after the activity is started, the leak will not exist.

Table 16: Static Variable Activity Leak Test Result in (KB)

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	54	0	0	0	0
Nexus 6 API 19	4.4	78	0	0	0	0
TECNO Camon CX API 24	7.0	57	0	0	0	0
Nexus 6P API 23	6.0	37	0	0	0	0
Nexus 5X API 17	4.2	46	0	0	0	0

6.2.6 Refactored Static Variable Activity Leak

The test results after removal of the static keyword yield the test result as follows in the table.

Table 17: Optimized Static Variable Activity Leak Test Result in (KB)

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	0	0	0	0	0
Nexus 6 API 19	4.4	0	0	0	0	0
TECNO Camon CX API 24	7.0	0	0	0	0	0
Nexus 6P API 23	6.0	0	0	0	0	0
Nexus 5X API 17	4.2	0	0	0	0	0

The test indicates that the static keyword is the main source of leaks in this pattern. As a result, the developers can save some amount of memory by removing such kind of references within the activity.

6.2.7 Singleton Object Activity Context Leak

Following the procedures in Figure 62, the leaks sizes are recorded in Table 11 for five tests.

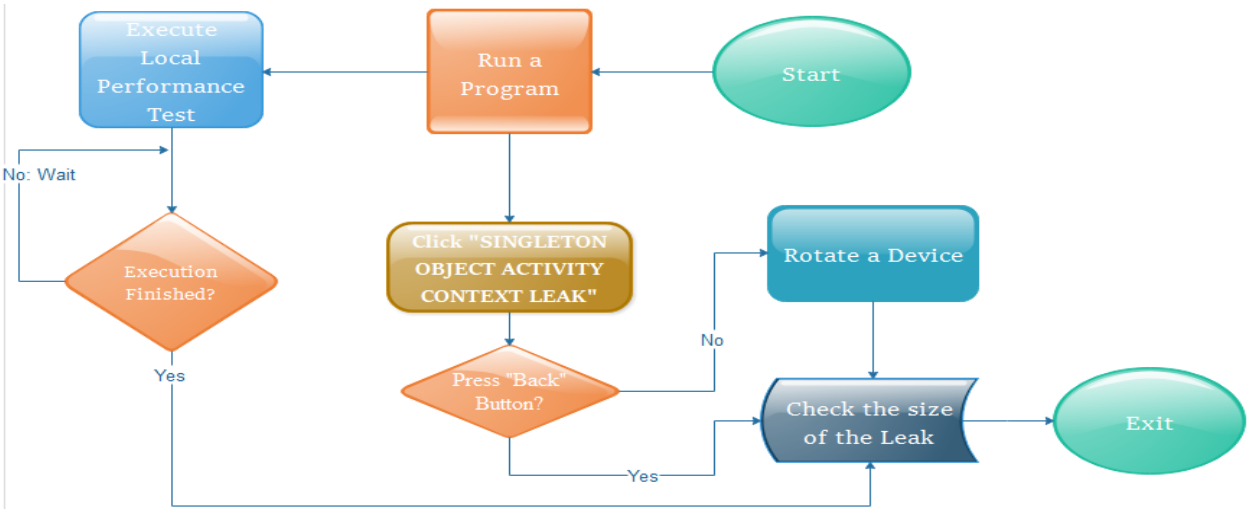


Figure 62: Leak Identification and Testing Flowchart for Singleton Object Leak Pattern

Even though this leak is not consuming a large amount of memory like long-running activities and static view context does, it leaks a memory for at least for the lifetime of the activity.

Table 18: Singleton Object Activity Context Leak Test Result in (KB)

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	54	0	0	0	0
Nexus 6 API 19	4.4	4.5	0	0	0	0
TECNO Camon CX API 24	7.0	57	0	0	0	0
Nexus 6p API 23	6.0	37	0	0	0	0
Nexus 5X API 17	4.2	4.6	0	0	0	0

The test result indicates that there is no leak after the activity is destroyed. On the other hand, it leaks a memory once the device configuration changes or receives certain interaction from the users to perform an activity and to return to the main UI.

6.2.8 Refactored Singleton Object Activity Leak

The test result after applying the optimization techniques in Chapter five is illustrated in the table below with both of these two ways.

Table 19: Optimized Singleton Object Activity Context Leak Test Result in (KB)

Device Name	*Target Android*	*Test1*	*Test2*	*Test3*	*Test4*	*Test5*
Nexus 4 API 24	7.0	0	0	0	0	0
Nexus 6 API 19	4.4	0	0	0	0	0
TECNO Camon CX API 24	7.0	0	0	0	0	0
Nexus 6P API 23	6.0	0	0	0	0	0
Nexus 5X API 17	4.2	0	0	0	0	0

6.3 Impacts of the Patterns on the Apps Memory Consumption

To show the impacts of each memory resource leak pattern on memory consumption of the running application that interacts with other system processes and threads the test result was recorded in the following three states:

- *Beginning state:* for each application starting time, the application startup memory size is recorded.
- *Click and Press back the Button or Rotating State:* there is a little interaction (6 times each) with the application. Here Garbage collector works frequently to reclaim the memory with plenty of ups and downs.
- *Persistent State:* Here, the interaction with the running app (clicking or rotation) is stopped and the GC is forced to start. In this state, Garbage collector is clicked many times until it cannot reclaim any memory. As a result, the memory that the running application is using stand permanently.

The size of the memory used in these three states is recorded, and then, the memory size of the beginning state is subtracted from the memory size of the persistent state. This result gives the amount of the memory leaked due to the specific memory leak patterns on the application. The effects of the patterns are shown in Table 18 (with an emulator of Android 7.0) and Table 19 (with real device TECNO Camon CX of Android 7.0).

Table 20: Memory Leak Patterns Impact on Memory Consumption of Running App Using Emulator

Memory Resource Leak Patterns	***State***			
	Beginning State (MB)	*Click/ Rotation State (MB)*	*Persistent State (MB)*	*Leak Amount (in MB)*
AsyncTask Activity	10.59	18.35	16.4	5.81
Thread Activity	10.45	23.03	16.8	6.35
Handler Activity	10.58	22.93	16.49	5.91
Static Variable Activity	10.58	16.82	14.89	4.31
Static View Activity	10.44	15.05	14.81	4.37
Singleton Object Activity	10.43	16.64	14.8	4.37
Inner Class Activity	10.4	15.12	14.82	4.42

As we see the results from the Table 19, the average memory resource leakage through long running tasks (6.02 MB) is larger than the average memory leakage through static reference leaks (which is 4.37 MB).

Table 21: Memory Leak Patterns Impact on Memory Consumption of Running App Using TECNO Camon CX

Memory Resource Leak Patterns	*State*			
	Beginning State (MB)	*Click/ Rotation State (MB)*	*Persistent State (MB)*	*Leak Amount (in MB)*
AsyncTask Activity	43.75	48.61	48.64	4.89
Thread Activity	44.16	48.67	48.79	4.63
Handler Activity	43.96	48.54	48.58	4.62
Static Variable Activity	44.04	46.4	46.34	2.34
Static View Activity	44.02	46.36	46.39	2.37
Singleton Object Activity	43.95	46.85	46.45	2.5
Inner Class Activity	43.58	46.48	46.42	2.84

The test result in Table 19 indicates that the average memory leakage of leak patterns in a real-time device due to long running task (4.71 MB) is larger than the average leakage resulting from static reference (which is 2.51 MB). Moreover, the result in click/rotation column is lower than that of persistent state for the long-running activity leak patterns and for the static view Activity leak. Because the leakage increases every time a new activity and view is created.

6.4 Threats to Validity

The result of the memory leak resource patterns impact on the memory of running application is found by either clicking and pressing the back button or rotating the view which is displayed by clicking a certain leak pattern six times. If there are lots of interaction with application throughout the time, then the indicated amount of memory leakage may increase especially leakage due to long-running activity and static view activity leak.

On the other hand, the evaluation of the impacts of leak patterns on running application is performed on ART runtime. Dalvik Virtual Machine (DVM) is not considered while testing the effect. In addition to this, the interaction with the application for the specific pattern is performed within two minutes.

CHAPTER SEVEN

CONCLUSION AND FUTURE WORK

7.1 Conclusion

The goal of this research is to automate the memory testing and to analyze the impacts of memory resource leak patterns on memory utilization of applications at the source code level by developing a benchmark application that consists seven leak patterns. The research process has seven phases: literature review, resource leak pattern identification, benchmarking, custom Gradle plugin development, performance evaluation test script development, perform instrumentation testing, code optimization.

In the first phase, literature review, test mechanisms, performance analysis, memory leak patterns, object references, Android and Android applications, Android memory management, and performance optimizations are thoroughly reviewed to articulate the problem and to show what has already addressed and what is not.

In the second phase, resource leak pattern identification, the total of seven patterns are identified with two categories: leaks through long-running activity and leaks through static references. The first category contains three leak patterns: AsyncTask activity leak, Handler activity leak, and Thread activity leak. Moreover, the second category consists of four memory leak patterns. These are Static view activity leak, Static variable activity leak, static variable to an instance of the inner class leak, and singleton activity leak.

The third phase is benchmarking, which is developing a sample application and using previously developed patterns that has the capability to show the previously identified resource leak patterns. In this phase, the major task is developing an application that has the activity which correctly shows the leakage properties of the patterns.

The fourth phase is custom Gradle plugin development, which can be imported and applied to a project Gradle build file for the continuous delivery of application development. This plugin helps for the automation of the test mechanism and execution of tests on every device that is connected

to the Android studio. It has also the capability of instantiating the test script via monkeyrunner to control the device outside of the code.

The fifth phase is performance evaluation test script development. To address the problems of analyzing the results which would be displayed on logcat, the test script is developed to move the logcat information that holds the test cases or activities information to the external directory. This creates an opportunity to trace details about the result of certain activities. This script is called within custom Gradle component with the help of monkeyrunner.

The sixth phase is performing an instrumentation testing, which helps to collect the instrumentation information. Junit4 test rules are used to manage the lifecycle of key application components involved in the test such as services and activities. This rule is supported by espresso test cases which simulate user's interaction to the activity or by performing an action on view objects. When the instrumentation testing finishes its execution, it is written on the external directory with full logcat and test failure (if any) information which then is used for the analysis of the impacts of memory leak patterns on memory consumption through certain actions.

The seventh phase is code optimization, which is performed based on the test results found after executing instrumentation testing. Garbage collection calls and certain reference to activities are considered before optimizing the program. Then, without altering activities functionalities, the best possible solution is proposed and applied to the leak patterns. After optimization, the instrumentation testing is performed to verify whether the proposed solution is worked or not.

From the analysis results of leak patterns impact on single running application memory utilization, the long-running activity leaks larger in both with the emulator and real device (average 6.02 MB and 4.71 respectively) than leaks through static reference (average 4.37 MB and 2.51 MB respectively). Moreover, the leakage in long-running activities and static view activity leak increases throughout the time continuously.

Generally, the study addressed all the research questions as follows:

RQ1: What are the issues that lead to inefficient memory utilization?

There are different issues that drive the applications to consume the abnormal amount of memory. Among these issues: strong reference to unused objects, creation of unnecessary objects, missing of releasing operations, programming languages that don't have built-in automatic GC such as C

and C++, programming styles, bad implementation practices such as failed invocation of memory related API class and lack of exception handling leads to inefficient memory utilization.

RQ2: How the test of memory utilization can be automated?

The memory testing can be automated by developing and defining custom tasks and methods in a custom Gradle files that will be synced with the project Gradle file for checking the code changes in a continuous delivery and defining a test cases that will simulate user's interaction to the application to expose the defined activities effect as well as by defining test scripts for extracting memory information to the external directory (to simplify the analysis).

RQ3: How the leak patterns affect the memory performance of the running application?

When the application that consists associated resource leak executes in the implementation devices and specific application framework events such as button clicking and activity lifecycle is performed during execution, then the amount of memory allocated will be increased. Because, while there is an interaction with an application, there is an interaction with plenty of system processes and other threads that will lead to high memory allocation. As a result, leak patterns lead the application to consume abnormal memory as it calls plenty of API that will be used by the running applications process and will not be released in some way till some conditions are met.

RQ4: How the memory utilization of android application can be improved by automated test of the memory?

We have found that the memory performance can be optimized by applying the following best practices:

- Instead of making classes non-static that references the outer class, making the class static or using a weak reference
- Declaring threads to static to prevent the implicit reference to the containing class
- Including the cancellation policy or unregistering objects at the onDestroy callback
- Making view object non-static because view objects will never get killed and
- Setting unused objects to null and removing static keywords from member variables such as static variables.

7.2 Future Work

Future work within this area could involve repeated measurement and analysis on devices that run future versions of Android if there are improvements in memory management. The areas that need further investigation in the future include:

- Consideration of DVM while evaluating memory leak patterns impact on the memory utilization of the running application.
- Enable the module-based memory leak analysis (combined memory leak pattern) instead of evaluating each components impact.
- Implementing the proposed framework and the best practices on the real-time applications that can be downloaded from App stores.
- Evaluating the proposed test automation effectiveness and efficiency with other testing.
- Finally, it would be desirable to increase the precision of memory leakage and management measurement.

References

[1] A. Lewerentz and J. Lindvall, *Performance and Energy Optimization for the Android Platform*, Blekinge Institute of Technology, 2012.

[2] D. Li and W. G. J. Halfond, "An investigation into energy-saving programming practices for android smartphone app development," in *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, 2014.

[3] M. Nagappan and E. Shihab, "Future trends in software engineering research for mobile apps," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, 2016.

[4] M. Linares-V{\'a}squez, C. Vendome, Q. Luo and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in android apps," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, 2015.

[5] A. Dogtiev, "App Download and Usage Statistics 2017," 21 November 2017. [Online]. Available: http://webcache.googleusercontent.com/search?q=cache:http://www.businessofapps.com/data/app-statistics/. [Accessed 28 November 2017].

[6] D. Moth, "85% of consumers favour apps over mobile websites," Econsultancy, 12 March 2013. [Online]. Available: https://econsultancy.com/blog/62326-85-of-consumers-favour-apps-over-mobile-websites. [Accessed 28 November 2017].

[7] E. Sherman, "The top 6 reasons mobile apps crash: How to best avoid Murphy," Techbeacon, 2015. [Online]. Available: https://techbeacon.com/top-6-reasons-mobile-apps-crash. [Accessed 28 November 2017].

[8] H. Fakhruddin, "Top 12 Reasons Why Users Frequently Uninstall Mobile Apps," Teknowledge, 5 January 2016. [Online]. Available: http://teks.co.in/site/blog/top-12-reasons-why-users-frequently-uninstall-mobile-apps/. [Accessed 28 November 2017].

[9] M. H. Memon, M. Hunain, A. Khan, R. A. Shaikh and I. Khan, "Power management for Android platform by Set CPU," in *Computing for Sustainable Global Development (INDIACom), 2016 3rd International Conference on*, 2016.

[10] A. Developers, "Platform Architecture," Google, [Online]. Available: https://developer.android.com/guide/platform/index.html. [Accessed 31 January 2018].

[11] A. Developers, "Application Fundamentals," Google, [Online]. Available: https://developer.android.com/guide/components/fundamentals.html. [Accessed 31 January 2018].

[12] A. Developers, "The Activity Lifecycle," Google, [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle.html. [Accessed 31 January 2018].

[13] 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*, 2014.

[14] G. Developers, "Understand the Activity Lifecycle," Google, 17 April 2018. [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle. [Accessed 2 May 2018].

[15] A. Developers, "Overview of memory management," 18 April 2018. [Online]. Available: https://developer.android.com/topic/performance/memory-overview. [Accessed 27 April 2018].

[16] K. Vimal and A. Trivedi, "A memory management scheme for enhancing performance of applications on Android," in *Intelligent Computational Systems (RAICS), 2015 IEEE Recent Advances in*, 2015.

[17] "Java Memory Management," Dynatrace, [Online]. Available: https://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works/. [Accessed 8 May 2018].

[18] K. Yaghmour, Embedded Android: Porting, Extending, and Customizing, " O'Reilly Media, Inc.", 2013.

[19] Y. Cheon and A. Escobar De La Torre, "Impacts of Java Language Features on the Memory Performances of Android Apps," 2017.

[20] M. Ju, H. Kim, M. Kang and S. Kim, "Efficient memory reclaiming for mitigating sluggish response in mobile devices," in *Consumer Electronics-Berlin (ICCE-Berlin), 2015 IEEE 5th International Conference on*, 2015.

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

[22] G. Developers, "Performance tips," 17 April 2018. [Online]. Available: https://developer.android.com/training/articles/perf-tips. [Accessed 1 May 2018].

[23] M. Jain and D. Gopalani, "Memory leakage testing using aspects," in *2015 International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2015.

[24] N. M. R. R. Geoffrey Hecht, "An Empirical Study of the Performance Impacts of Android Code Smells," in *ACM International Conference on Mobile Software Engineering and Systems*, 2016.

[25] A. Sinhal, "Avoid Memory Leaks and understanding of reference in Android," A Medium Corporation, 28 January 2017. [Online]. Available: https://medium.com/@ankit.sinhal/avoid-memory-leaks-reference-afd0e9dbb213. [Accessed 17 February 2018].

[26] T. Das, M. Di Penta and I. Malavolta, "A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, 2016.

[27] M. Linares-V{\'a}squez, C. Bernal-C{\'a}rdenas, K. Moran and D. Poshyvanyk, "How do developers test android applications?," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, 2017.

[28] A. Abogharaf, R. Palit, K. Naik and A. Singh, "A methodology for energy performance testing of smartphone applications," in *Proceedings of the 7th International Workshop on Automation of Software Test*, 2012.

[29] D. Yan, S. Yang and A. Rountev, "Systematic testing for resource leaks in Android applications," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, 2013.

[30] W. Oliveira, R. Oliveira and F. Castor, "A study on the energy consumption of Android app development approaches," in *Proceedings of the 14th International Conference on Mining Software Repositories*, 2017.

[31] T. Wu, J. Liu, X. Deng, J. Yan and J. Zhang, "Relda2: an effective static analysis tool for resource leak detection in Android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[32] J. Liu, T. Wu, J. Yan and J. Zhang, "Fixing resource leaks in Android apps with light-weight static analysis and low-overhead instrumentation," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, 2016.

[33] S.-j. K. M.-s. J. H.-J. Yoo, "Study of Garbage Collection Performance on Dalvik VM," in *International Conference on IT Convergence*, 2013.

[34] S. Hamanaka, S. Kurihara, S. Fukuda, M. Oguchi and S. Yamaguchi, "Application state aware GC selection optimization in Android," in *Consumer Electronics-Taiwan (ICCE-TW), 2016 IEEE International Conference on*, 2016.

[35] G. Lim, C. Min and Y. I. Eom, "Enhancing application performance by memory partitioning in Android platforms," in *Consumer Electronics (ICCE), 2013 IEEE International Conference on*, 2013.

[36] F. Alam, P. R. Panda, N. Tripathi, N. Sharma and S. Narayan, "Energy Optimization in Android Applications Through Wakelock Placement," in *Proceedings of the Conference on Design, Automation \& Test in Europe*, 3001 Leuven, Belgium, Belgium, 2014.

[37] A. Carette, M. A. A. Younes, G. Hecht, N. Moha and R. Rouvoy, "Investigating the energy impact of android smells," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, 2017.

[38] X. Li and J. P. Gallagher, "A source-level energy optimization framework for mobile applications," in *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, 2016.

[39] Z. Z. X. Chen, "Android App Energy Efficiency: The Impact of Language, Runtime,Compiler and Implementation," in *IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)*, 2016.

[40] D. G. D. P. a. S. G. A. Ferrari, "Detecting Energy Leaks in Android App with POEM," IEEE International Workshop on the Impact of Human Mobility in Pervasive Systems and Applications, 2015.

[41] C. W. S. Q. Z. Xu, "State-taint analysis for detecting resource bugs," Science of Computer Programming, 2017.

[42] A. Banerjee, L. K. Chong, C. Ballabriga and A. Roychoudhury, "Energypatch: Repairing resource leaks to improve energy-efficiency of android apps," *IEEE Transactions on Software Engineering*, 2017.

[43] N. Ding, D. Wagner, X. Chen, A. Pathak, Y. C. Hu and A. Rice, "Characterizing and modeling the impact of wireless signal strength on smartphone battery drain," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, pp. 29-40, 2013.

[44] S. Rosen, A. Nikravesh, Y. Guo, Z. M. Mao, F. Qian and S. Sen, "Revisiting network energy efficiency of mobile apps: Performance in the wild," in *Proceedings of the 2015 Internet Measurement Conference*, 2015.

[45] M. A. S. K. C. J. Y. a. K.-Y. J. Jung-Hoon Shin, "Examining Performance Issues of GUI Based Android Applications," in *Advanced Multimedia and Ubiquitous*, 2016.

[46] S. Yang, D. Yan and A. Rountev, "Testing for poor responsiveness in Android applications," in *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the*, 2013.

[47] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann and D. Lo, "Understanding the test automation culture of app developers," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, 2015.

[48] G. Santhanakrishnan, C. Cargile and A. Olmsted, "Memory leak detection in android applications based on code patterns," in *Information Society (i-Society), 2016 International Conference on*, 2016.

[49] H. Fakhruddin, "Eclipse vs Android Studio: Which IDE Is Better For Android Developers?," 10 March 2015. [Online]. Available: http://teks.co.in/site/blog/eclipse-vs-android-studio-ide-better-android-developers/. [Accessed 17 January 2018].

[50] N. Minaev, "The Top 5 Android UI Frameworks for Automated Testing," DZone, 17 September 2017. [Online]. Available: https://dzone.com/articles/the-top-5-android-ui-frameworks-for-automated-test. [Accessed 23 January 2018].

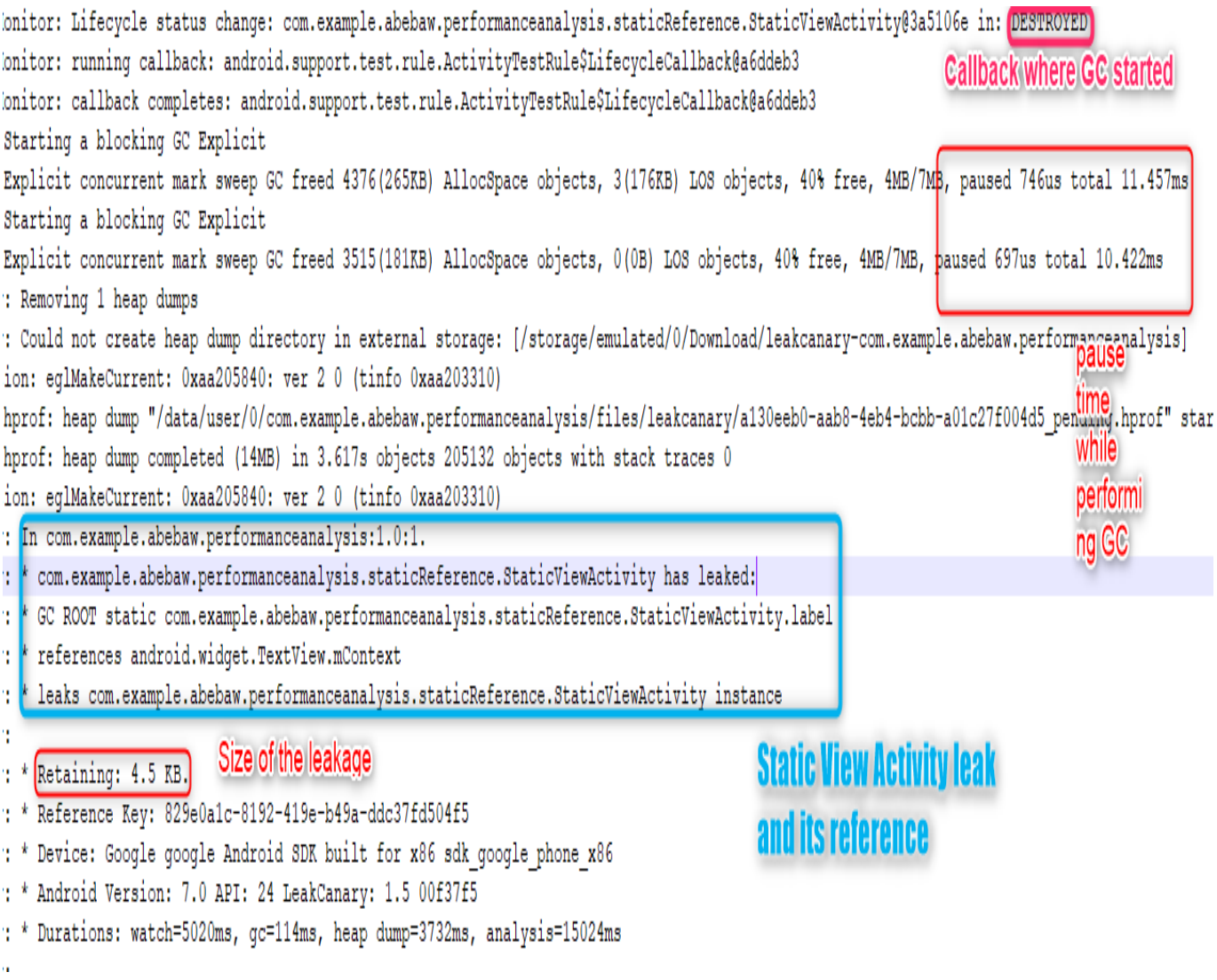
[51] G. Developers, "Distribution dashboard," 7 May 2018. [Online]. Available: https://developer.android.com/about/dashboards/. [Accessed 15 May 2018].

[52] A. Goransson, Efficient android threading: Asynchronous processing techniques for android applications, " O'Reilly Media, Inc.", 2014.

Appendixes

Appendix I: Static view activity leak logcat File retrieved through automated testing

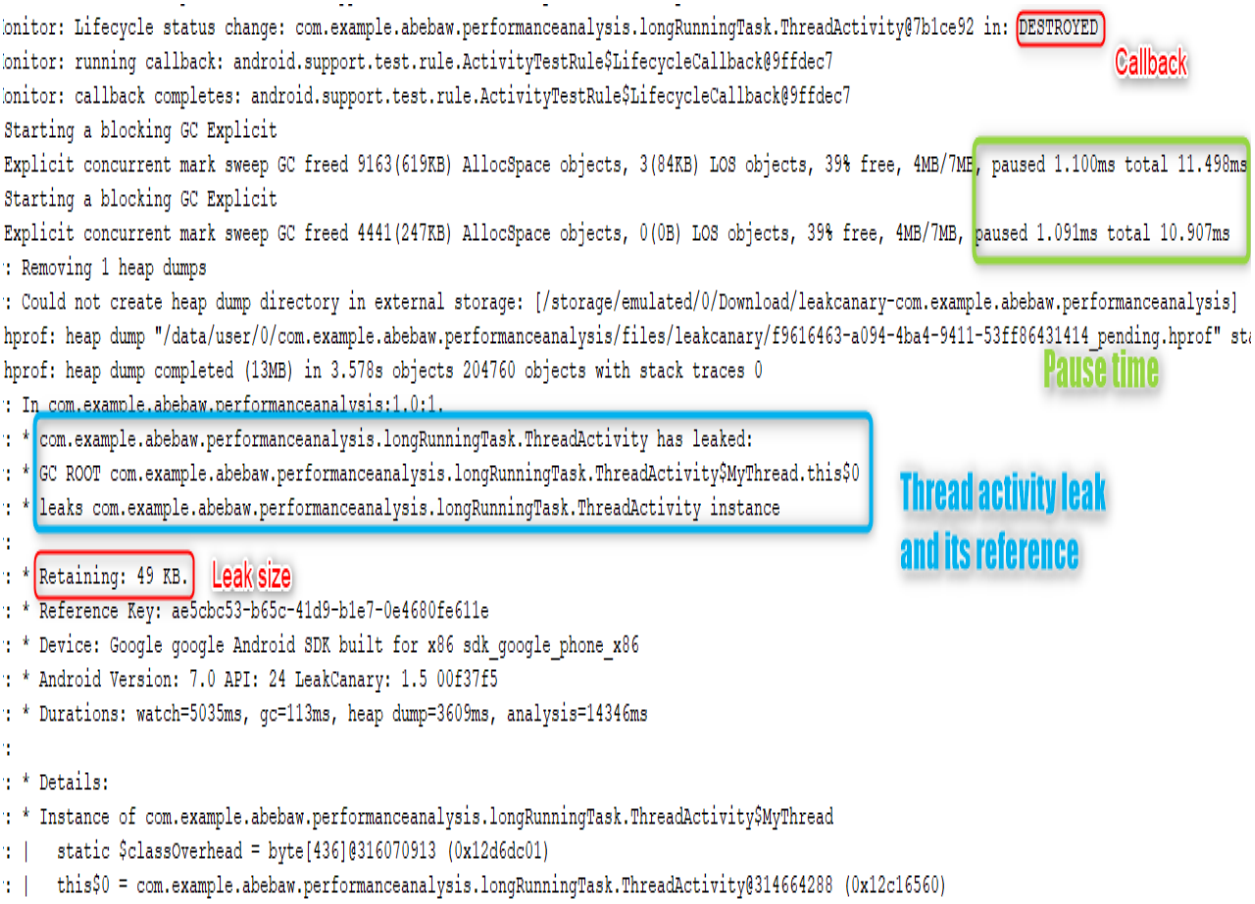
Logcat file retrieved through an automated test to the external directory for static view activity leak memory leak pattern:



In the same way, the rest of three static reference leak pattern output is extracted to the external directory in this format. As a result, to not make it redundant, the rest leak pattern logcat information is not included here.

Appendix II: Thread activity leak logcat File retrieved through automated testing

Thread activity leak from the long-running activity task is extracted the overall activity information in logcat in the following way:



AsyncTask activity leak, and Handler activity leak logcat information is not also included here in the appendix as it will be redundant. Because, the only thing they differ with this one is the amount of garbage collection calls (if any), leaking reference, and pause time.