Journal of Theoretical and Applied Information Technology
30th November 2020. Vol.98. No 22

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ENERGY-LEAKS IN ANDROID APPLICATION DEVELOPMENT: PERSPECTIVE AND CHALLENGES

MUHAMMAD UMAIR KHAN[1] , SHANZA ABBAS[2], SCOTT UK-JIN LEE[3*], ASAD ABBAS[4]

[1,2]Departement of Computer Science and Engineering, Hanyang University, Republic of Korea

[3]Departement of Computer Science and Engineering, Major in Bio Artificial Intelligence, Hanyang University,

Republic of Korea

[4]Faculty of Information and Technology, University of Central Punjab, Lahore, Pakistan

mumairkhan@hanyang.ac.kr, shanza92@hanyang.ac.kr, scottlee@hanyang.ac.kr*, asadabbas.grw@ucp.edu.pk

ABSTRACT

Number of mobile devices are increasing, and popularity of Android OS supported devices is more than other, and Android apps dominated the mobile apps market. These devices have limited resources (CPU, Memory, and Power, etc.). Energy consumption in mobile devices is an important factor to consider when developing an app. There is no such official guide that help developer to build a less energy-hungry app or discover energy leaks in the development phase. This paper will review different types of energy leaks in Android apps, how these leaks effect the energy consumption of the device. We will discuss how these energy leaks can be avoided at the development phase

Keywords- *Energy Leaks, Code Smell, Android Apps, Energy Consumption, Mobile Apps, Program Analysis*

1. INTRODUCTION

In recent years mobile devices are used abundantly. Peoples are mostly using mobile apps for their basic computing. Mobile apps are new emerging mainstream of software system which presents numerous challenges for the researcher in software engineering[1]. There are almost 2.7 million android apps and 2.2 million iOS apps available online and the average use of mobile phone per person is 2 hours and 51 minutes in a day [2]. Android apps dominated the app market of mobile, because of the popularity of open-source OS and a variety of free apps available on Play Store [3].

Mobile devices come with limited resources such as computing, memory, and power. They are continuously growing in size and complexity. Mobile apps use resources which include animation, more color depth in the screen, accurate sensor data, memory, and CPU power. Accessing these resources in an app will consume energy. Mobile devices cannot have larger battery because they use Lithium batteries which are lightweight and portable but has limited battery capacity. Energy consumed by mobile apps plays a vital role as the use of the mobile phone is increased so the user requires more battery life. Apps need to reduce energy consumption as much as possible to facilitate users [4], [5].

Android app developers only use best practices (provided by Google's best practices) to create the application. New or inexperienced developers try to archive only the functional requirements. They ignore non-functional requirements, especially energy consumed by the app. Adequate energy measurements are also not well supported at the time of development, as there is not any official energy measuring tool. Google published a set of best practices to optimize the performance [5], [6], [7], but these practices does not provide the guideline to reduce energy consumption [9].

Mobile devices contain different sensors (e.g. Wi-Fi, GPS, Proximity, etc.) that app uses to achieve the functionality they provide to the user. Mobile apps use the data from the sensor, which is used by a mobile app to help user accordingly, for example, GPS provides location update that can be

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

used in maps to find nearest attractions. Developer should use these sensors carefully because extensive use or not properly closed resources will result in the drain of battery. To access these resources, we need APIs. Table 1 shows some hardware resources with their system call API, that mostly consume energy in mobile devices.

API's are used to assist developers, for fast and easy access to resources in Android apps, so they play a vital role in the development of the Android application. Basic API's are provided by the Android OS but still, there are many third-party APIs that are used in the development [10]. These APIs provide different methods to access resources with different properties. Lack of knowledge of method working will lead to unexpected results and may terminate the program on some circumstances. For example, most of the apps use the internet to communicate, they use HTTP calls API to communicate. These calls usually contain a small amount of data, but to communicate with the server, they need to enable data connection or Wi-Fi (which may be in sleep state). When they are finished communicating, these sensors will go to idle or sleep. The additional energy consumed after communication and before sleep is called "tail energy". HTTP calls can be optimized by bundling a small HTTP request [10]. Similarly, the free app contains advertisements, which consume bandwidth and screen active time causing battery consumption.

Table 1: Android System Call Apis That Influence Energy Consumption

Resources	System Call APIs	Hardware
LocationManager	requestLocationUpdates / removeUpdates addProximityAlert/ removeProximityAlert	GPS
WifiManager	setWifiEnabled acquire/release	Wi-Fi
Camera	open/close startPreview/stopPreview	Camera
Sensor	registerListener/ unregisterListener	Gyro, pedometer, proximity and so on.
LocationClient	requestLocationUpdates / removeUpdates addGeofences/removeGeofances	GPS
MediaRecorder	start/stop	Audio/video
BluetoothAdapter	enable/disable	Bluetooth
CellularData	enable/disable	GSM

Different studies try to solve energy leaks in Android applications, but these studies only limited to detect and fix one problem or few problems [9], [10], [11], [12], [13]. They leave many unattended energy leaks. There is no proper guide or tool to detect all the energy leaks or to fix all the problems related to energy in apps. These energy leaks can be small or big. If they are small, they may go unnoticed by user and developer. But if they leak a large amount of energy then the user of the app reports this bug. As the developer only knows about energy leaks from user feedback and there is no official tool to assist them in detecting these energy leaks. This motivates us to study and find different energy leaks in mobile apps.

This paper provides a review of different energy leaks presented by different studies and discusses how a simple mistake can affect the energy consumption of the Android apps. We then discuss the solutions to avoid these energy leaks when developing the apps.

The paper is organized in the following manner: Section-2 will give some basic background, Section-3 will review related works. Section-4 will discuss different energy leaks. In Section-5 we see how these leaks can be avoided and Section-6 will conclude this work.

2. BACKGROUND

To find energy leaks in Android apps we need to know how the applications are collected, executed and analyzed. This section will give a short overview of how we can gather Android applications to have conclusive results. How application runs during their life period i.e how they are initiated, executed and destroyed. Then we will see what the different options are to analyze these applications. We will also see the effect of best practices provided by Google on energy consumption.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

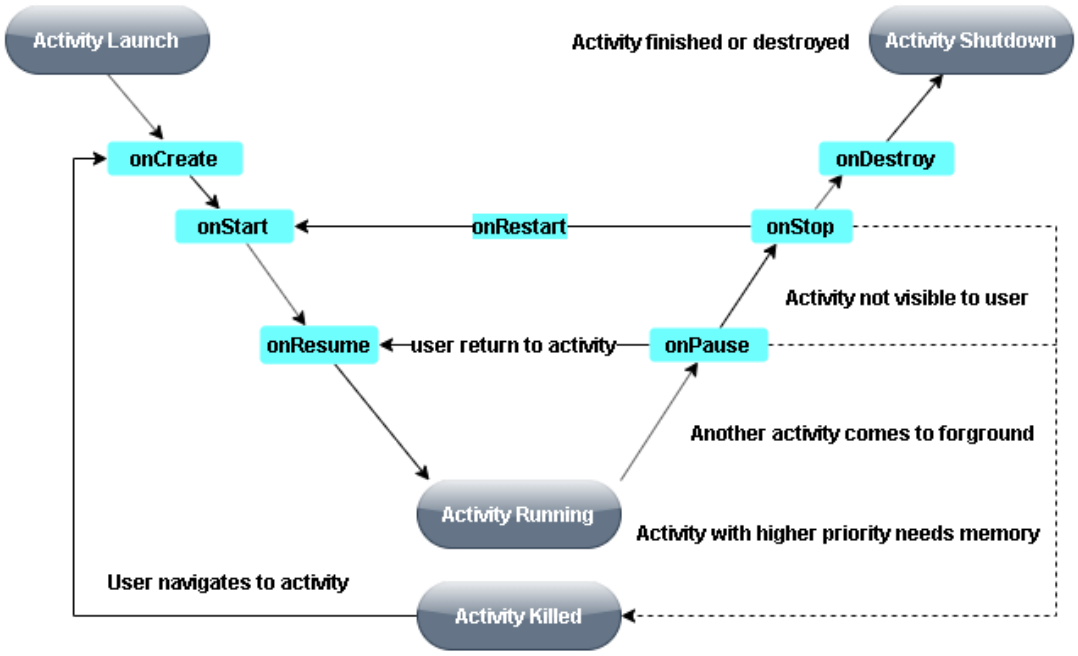

Figure 1: Android Activity Lifecycle

2.1. Gathering Dataset

To analyze energy leaks in apps we need to have a set of apps that need to be analyzed for energy leaks. Gathering open-source Android app is a time-consuming task and not all the apps have source code available. For the apps that do not have source code, they need to be analyzed statically by converting their APK file into Dalvik code and then making control flow graph for further processing. We need to have a dataset which needs to be analyzed to give more in-depth knowledge and conclude the results. These datasets can be downloaded from AndroZoo, Android Malware Genome Project, Google Play, F-Droid, appannie, appbrain, appszoom, AppChina, Anzhi, AndroVault, and apkmirror [15]. To analyze apps for energy consumption these stores are used in different studies.

2.2. Android Application Lifecycle

Android apps are different from conventional java applications. They do not have any main method (which is executed at the start of the program), they use event-driven model and generally revolve around a GUI so we cannot predict the flow of the program. Android applications are composed of different types of component:

- *Activity* is the entry point of application to interact with the user, which can be composed of a series of independent and collaborative activities.
- *Service* perform long-running operations that runs in the background.
- *Broadcast receiver* is used to communicate with different system events as it responds to system-wide broadcast messages. It works as a gateway to other components.
- *Content provider* manages a shared set of application data i.e. other components and applications can query or modify the data[16].

There are seven distinct stages of the lifecycle of an activity when it is created as shown in *Figure 1*, at the beginning of the activity execution *onCreate*, *onStart* and *onResume* are invoked. All tasks related to initialization and resource acquisition are usually performed in these stages. Similarly, when stopping the execution of the activity *onDestroy*, *onStop* and *onPause* are invoked. *onRestart* method is used to restart previously stopped activity. To implement custom functionality developers, need to override these methods[11].

Android framework defines thousands of callbacks for different interaction. These interactions are activities, menus, and dialogs. There are two main categories of a callback. Lifecycle callbacks manage the lifecycle of the application. *onCreate* and *onDestroy* are significant because

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

they manage the activity lifecycle which is essential for developers. GUI event handler callbacks respond to user actions on the GUI (e.g.clicking button). They are called when the user interacts with GUI by clicking, they control the flow of the program. As Android devices are event-driven, events are the input, which can be either from user interaction or from the system[17].

2.3. Analyzing the Android App

To analyze the Android application, there are two types of analysis techniques: *static analysis* and *dynamic analysis*. The *static analysis* takes source code and examines this code without executing it, it gives an abstract model of the program e.g. call graph, control flow graph[14], [15], [16], [17], [18], [19], [20]. The main advantage of static analysis is that all the code is analyzed. *Dynamic analysis* monitors the execution of the app by inspecting the runtime behavior of the app. GreenDroid is a dynamic analysis tool to detect energy defects in Android apps [25]. The main advantage of dynamic analysis is that it performs analysis at runtime, depending on the scenario it provides the energy consumption of energy greedy part of the app which is helpful to reduce energy at that part of the program. Then there is a hybrid analysis that uses both static and dynamic analysis[21], [22], [23], [24], [25], [26]. These techniques are mostly used for security analysis of mobile applications[1]. As this paper is about the energy consumption of Android apps so we will discuss energy leaks in Android apps.

2.4. Effect of Best Practices

Android developer guide provides Google's best practice for Lint, which was analyzed and then measures the energy according to the priority in Lint [32], [33]. They found these practices have little effect on energy improvement although they are good for performance. They show that most energy-efficient practices may not give the best performance. They show that 21% of the dataset contains *ObsoleteLayoutParam* (i.e. layout no longer in use in UI), 16% contain Recycle (missing *recycle*() calls TypedArray objects should be efficiently used as they are a singleton). Result also shows that 32% of apps contain at least one energy inefficiency. We will discuss them in Section-4 and how we can improve them so that energy consumption is reduced.

3. RELATED WORK

There are different researches on detecting energy leaks of the mobile phone. This section will discuss some of the recent research.

Program analysis to detect energy Inefficiencies. To detect energy inefficiencies, Android apps need to be analyzed. Most of the research uses program analysis to detect inconsistencies [14], [19], [19], [27], [30], [32], [33]. *Alireza et al.*[35] classify and characterize research efforts in this area. They have provided the in-depth knowledge of program analysis and showed, what are the choices, which types of analysis, what is the relation, and where to apply which analysis. They showed that program analysis is mostly used to detect vulnerability and malicious apps. This will help the new researcher to help in exploring the current trends in Android apps analysis. Most of the research uses static analysis to detect energy defect [6], [11], [12], [22], [27], [34], [35], [36], [37], [38], [39], [40]. Some uses dynamic analysis[37], [45] and some uses combination of both also called hybrid[20], [22], [23], [42].

Resource management. Resource management in android app is an important task many of recent research are going on optimizing resource to improve the energy consumption of apps. Resource safety policy checking is adopted by Relda to detect resource leaks in the apps [34], [38], [43]. They assume acquire and release points, so cannot properly handle wake lock misuses in real-world apps. *Reyhaneh et al*. use the energy-aware mutation to find energy leaks [30]. They define the mutation operator to check if they found any energy leak by introducing mutant. Testing these mutants is a time-consuming process and most of the parameter are set by developers to get the work done in time so changing them may affect the flow of the program. *Anway et al*. consider the split-screen functionality of mobile, which actively run simultaneous applications and consume more energy [48]. They archive energy-efficient configuration by adaptively adjusting voltage and frequency of CPU and memory bandwidth. Changing the hardware parameter may change the behavior of other apps so it is not recommended. Some other work also suggests that resources used by the application need to be closed properly otherwise they will drain the battery [17], [29].

Energy-aware test generation. Generating an energy-aware test in the mobile application is a difficult task as Android official guide do not provide any guideline. There is no official tool available to measure energy consumption at development time. Developer need to use adb (Android Debugging Bridge) command to get the information of executing programs and device resource information. Test generation also requires measuring the energy consumption of the test. Researchers have proposed several techniques that generate test cases to detect and optimize energy

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

leaks in mobile apps [10], [16], [25], [46], [47], [48].

4. ENERGY LEAKS AND SOLUTIONS

This paper focuses on energy leaks in Android apps. Energy inefficiencies can be categories into *energy bugs*, which causes excessive energy consumption even after the app has finished execution and *energy hotspots*, which causes excessive energy consumption while the app is under execution. A literature review shows that energy consumption is the fourth most concerns area addressed by publication nowadays [1]. To improve the energy efficiency of mobile devices a catalog of 22 design patterns was constructed [52]. They use GitHub comments in the commit of Android app project and search for energy-related words such as energy, power or battery in comments. If they found any of these words, they first read the full comment and if it is energy improvement or bug identified they include it in the study. Fixes are included in the study by exploring their code changes. Then they construct their own catalog of energy leaks. Another study shows that four code smell types (internal setter, leaking thread, member ignoring method and slow loop) consume up to 87 times more energy than methods affected by other code smells. This study ignores other important code smells, that also affect energy consumption [14]. Energy leaks can be found by measuring the energy consumption of the device when running the app. This section will first give some overview of how we can measure energy consumption. Then we will discuss energy leaks and their solutions.

4.1. Measuring Power Consumption

After analyzing and detecting energy leaks they need to measure how much energy can be saved by removing the energy leaks. We can measure energy based on hardware and software approach. Hardware approach is more accurate but require extra hardware setup cost. Whereas, the software-based approach is less accurate but will show the power consumption of the app quite easily [10], [12], [13], [49]. Overview of some testing tool is shown in [16]. Software-based, eLens, is a light-weight tool to estimate the energy consumption of app at the level of fine granularity (i.e.for whole application, method, path, or source code line) [53]. Trepn[50], Petra[13] , eProf [54], and PowerTutor [55] are some examples of software-based energy measuring tools. To measure energy consumption using hardware, require special hardware to be set up. Although the measurements are accurate, it requires special hardware setup which cost extra. Some examples of hardware-based energy measuring tool are power meter, project volta[13], WattsOn [56], and Monsoon[13]. At the time of development developer only need to locate energy leaks, and how much energy is consumed by the program, so the software-based approach is helpful. It is also fast and requires fewer changes.

4.2. Energy Leaks

This section will describe the energy leaks found in Android OS. These energy leaks are detected by different studies. In this paper, we will summarize these leaks to get the overview of energy leaks. These energy smells play a big role in the energy consumption of mobile apps. The developer needs to minimize the leaks of energy in their apps.

4.2.1. Resource Leaks

Resource leaks are the main causes of energy leaks in mobile apps. They occur when some resources are acquired by the application during execution. But developer forgot to release that resource when exiting the application. These acquired resources must be released before exiting otherwise they will continue to be a high-power state.

When a service component starts, some system resources need to be acquired for later computation. **Error! Reference source not found.** gives an example of a library service [57]. The LibraryService component of the E-book reading app FBReader opens a database connection when it is launched (*Line* 5). This acquires database instance for interacting with the database (such as insert, delete operations). Resources should be released when the service is destroyed. Since acquiring and releasing operations are in different callback

```
1  public class LibraryService extends Service {
2      public void onCreate() {
3          ...// some set up work and
4          ...// open database
5          myDatabase - SQLiteBooksDatabase.Instance(...);
6      }
7      public void onDestroy() {
8          ...// some tear down work
9          myDatabase.close()
10     }
11 }
```

Figure 2: Example of Resource Leak

methods, developers can easily forget to release the resources properly. We can release the database connection using close (*Line* 9). Adding this will now remove the resource leak from the program. Similar can be the case of acquiring any Android service and forgetting to release it will cause a resource leak.

4.2.2. SensorLeak

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The mobile device came with multiple sensors such as camera, GPS, fingerprint sensor, accelerometer and so on. These multiple categories of sensors are represented by an integer constant. For example, *Sensor.TYPE_ACCELEROMETER*, which is a sensor object. If an application acquires a sensor during their lifetime but does not release it will cause the energy leak [58]. Hardware components/power management utilities can only be accessed by an app through a predefined set of system call APIs [12].

Error! Reference source not found. shows a simplified example [58] of sensor leak. To obtain sensor data, sensor *eventlistener* is registered by programmer *SensorEventListener* (*Line* 6). Whenever new sensor data is available callback *onSensorChanged* is invoked on this listener. As (*Line* 12) shows how a listener is registered with a sensor object. The sensor hardware will be enabled when any listener registered to listen to the sensor's changes. The listener will be removed using *unregisterListener* (*Line* 8) whenever the programmer wants to turn-off the hardware. The example shows the sensor leak when *UnlockActivity* is opened, the user may toggle *sc's* switch (state of switch *Line* 10-13) in the UI. It will invoke

```
1  class UnlockActivity extends Activity {
2      onCreate(...) {
3          SwitchCompat sc = ...;
4          SensorManager sm = ...;
5          Sensor accel = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
6          SensorEventListener shakeListener = new SensorEventListener {
7              onSensorChanged(...){
8                  if(...){sm.unregisterListener(this);}}};
9          sc.setOnCheckedChangeListener(new OnCheckedChangeListener){
10             public OnCheckedChangeListener(View v){
11                 ...
12                 sm.registerListener(shakeListener,accel);}}};
13     }
14     onDestroy() {...}
15 }
```

onCheckedChanged and as a result, will

- register shakeListener's listener (*Line* 12) object with accel's sensor object, and
- will wait for a shake gesture to unlock vault by the user.

Whenever the device is moved, *onSensorChanged* (*Line* 7) is invoked with information about the physical movement. If this movement is above some threshold (checked at *Line* 8), it is considered to be "shake to unlock". This will release the listener via *unregisterListener* and unlocks the vault. If the user quits this app and makes the phone stationary, *UnlockActivity* will be closed. At that time, lifecycle callback *onDestroy* (*Line* 14) does not release the sensor. Thus, the window that acquired the sensor does not release it and keeps the sensor alive. This will drain the battery. This example shows that the sensor will still be alive after the user quits the app, as the application process remains active upon quitting. Table 1 shows a different sensor with system API calls. These sensors should be carefully registered (such as SensorEventListener) and whenever the information from the sensor is not required, they should be unregistered (such as unregisterListener).

Figure 3: Example of Sensor Leak

4.2.3. Wake lock Misuses

Wake locks enable developers to explicitly control the power state of an Android device. Developers need to declare the *android.permission.WAKE_LOCK* permission in their app's manifest file [59]. Creating a *PowerManager.WakeLock* instance, and specify its type (partial, screen dim, screen bright, full, proximity screen off). Each type has a different wake level and different effect on energy consumption. For example, full wake lock will keep device CPU running, and screen and keyboard backlight on at full brightness. After creating instances of wake lock, certain APIs can be involved to acquire and release wake locks. Once wake lock is acquired, it will have long-lasting effects until it is released, or the timeout expires at a specified time. As wake locks directly affect device hardware state, and to avoid undesirable consequences developers should carefully use them [12], [14]. We will give an overview of 8 patterns of wake locks.

- *Unnecessary wakeup*: Developer acquired wake lock too early or released it too late which causing unnecessary wake-up time.
- *Wake lock leakage*: Once acquired wake lock it should be released on all program paths in android app.
- *Premature lock releasing*: It occurs when the wake lock is released before being acquired, which may cause the app to crash.
- *Multiple lock acquisition*: As wake lock is referenced counted and each acquisition of wake lock will increment the counter. All the acquired locks should be released as wake lock will only be released when the counter reaches zero[59].
- *Inappropriate lock type*: Developer should acquire the lock of the hardware that he wants to use or prevent from sleep. For instance, when using Wi-Fi for calls it should lock Wi-Fi, not CPU as Wi-Fi will go to sleep after some time and call will disconnect.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

- *Problematic timeout setting:* when acquiring a lock, developers can set a timeout. After that timeout, the acquired lock will automatically be released. Value of timeout is critical because it should not be too short or too long.
- *Inappropriate flags*: When acquiring wake locks, developers can also set some predefined flags. For example, ON_AFTER_RELEASE flag will cause the device screen to remain ON after wake lock is released. So, they should be carefully used to avoid undesirable consequences.
- *Permission Error:* App requires to declare the android.permission.WAKE_LOCK permission when acquiring wake lock. Forgetting to declare will lead to security violation and may crash the app. The developer must declare permission if the app will acquire the lock.

4.2.4. Context Leak

An object has a life cycle and context is the bridge between components. It is used for communication between components, instantiate components and access components. When context has reached the end of its life cycle but is still reachable from running thread or from the static field is known as context leak [38]. Examples of Android context are activities, services, broadcast receivers and fragments (the portion of the user interface) which are not contexts but contain a context. These context containers can generate context leaks. For example, when an activity is invoked it runs in foreground and previous activity loses the focus and goes down in a stack. The user interacts with the activity in the foreground only. Another activity is kept in the stack until invoked and come to the foreground. The lifecycle of Android activity is shown in **Error! Reference source not found.**. There are three types of context leaks found in practice:**Error! Reference source not found.**

- *Thread*: Threads are the long-running task (such as network and database operations) which run in the background to preserve the responsiveness of the application [14]. They should be stopped properly so that they can be removed from memory [38]. Thread related classes in Android are Thread, Runnable, Handler, HandlerThread and AsyncTask Thread running apps should stop the thread when they are destroyed or paused by the user. Unable to stop the thread will cause unnecessary computation energy.
- *Static field*: Programmers use static fields as global access points for shared data. They belong to a class, not to a specific instance[38]. If a programmer store context or context container in a static field (possibly indirectly). It might get leaked because it will be there till the program is running. A solution would be to reset the static fields to null which need to be implemented in the onDestroy() method.
- *System callback*: Android has a set of managers to interact with OS. They are created through static fields using Context.getSystemService() method. For example, location manager allows one to query the device location, the sensor manager allows access to sensor data (such as GPS). If callback object gets attached to managers to listen to specific event they will remain allocated until explicitly detached. A solution to this problem is to unregister the callback in onDestroy() method.

4.2.5. Long-wait State

If activity adds a listener callback and after a while, the user presses HOME, Back or Power button, the activity will send in the background and might not come to the foreground. Thus the activity is in a long-wait state if the listener is not removed [11]. For instance, callback *onLocationChanged* is called by an application to read location but after some time user presses HOME or POWER button. The application will go to the background, but listening is still active. Which will drain the battery, as the location is not required by the application. To solve this problem developer should remove the listener in *onPause*() or *onDestroy*() method

4.2.6. Data Transmission without compression

Almost all Android apps use the internet for communication. This communication could be through Wi-Fi or cellular this requires enabling the hardware. The enabled hardware was in idle state (low power state) but now it is in enabled state (high power state) which consumes more energy. Most of the time request are of small size and when these requests are sent and received, they take less time, but hardware will stay enabled for longer time causes extra energy consumption. Similarly, Ads of free apps consume network data. These request could be bundled together or compressed to save the transmitting power[14].

5. AVOIDING ENERGY LEAKS

We have discussed the energy leaks in Android apps that will lead to energy inefficient app. Developers should consider the energy

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

efficiency of the device and should avoid these energy leaks. They will lead to undesirable energy consumption.

These leaks are mainly due to resources acquired by the developer are not released or released at an inappropriate place which should be avoided. Control flow graph, event flow graph, and data flow graph can help the developer to get the insight into the flow of the program. These graphs are constructed using static analysis. If there is a missing released resource developer should add release. If there is a release but maybe at an inappropriate position, then make sure that it is released before the app closes. Developer should monitor energy-hungry resources (such as GPS, Wi-Fi, cellular network, and display). If it is consuming unnecessary energy, then it should be minimized by putting the resource in sleep when it is not active.

Android API provides a different parameter for hardware resources. They can be used by developers to minimize the usage of hardware resource. The hardware should only be used when its data is needed. For example, for GPS we can set the priority PRIORITY_HIGH_ACCURACY or PRIORITY_BALANCED_POWER_ACCURACY depending on the requirements. Similarly, we can *setInterval()* of updated values. If the interval is small, then the value of the sensor will be updated after a short time and consume battery. Developer should adjust these parameters according to the requirements of the app.

Context plays an important part when using the sensor. For example, if a user is not traveling fast (such as walking or at rest) he doesn't require the frequent update of position, but if he is traveling fast (such as on a car) then he needs fast and accurate updates. Developer should also decide whether to stop the sensor update or not if the app is send to background by user and user is not currently using the app.

Thread and wake locks are also the cause of energy leaks. Threads are working in the background (to get data from network or sensor) and need to be stopped when the work is complete or when closing the program. We have discussed wake locks so they should also be released when they are not required, and the timeout value should be carefully set by the developer.

Developer should also clear memory if they use static fields so that it can no longer consume memory by setting them to NULL. They should also consider that user may press HOME, BACK or POWER button to do some other task. As the task is switched, the app should release the resources to reduce energy consumption.

Loop is an important part of any program and duration of loop decide the performance of the program [14]. Android has two versions of loops and standard version of for loop is slower than for-each loop [20], [60]. Developers should always use the enhanced version of loops and other recommendations that improve the performance of the app.

6. CONCLUSION AND FUTURE WORK

Android apps are dominating the mobile apps market as they are mostly open-source, so most of the mobile devices use Android OS to cover a large group of users. Mobile devices have limited battery capacity so the apps running on these devices need to be less energy-hungry. This paper provides a review of energy leaks in Android apps, that causes unnecessary energy consumption in mobile apps. These leaks are mostly resource leaks, context leaks, sensor leaks, and wake lock misuse. To avoid these energy leaks, we also discuss the solutions that need to be considered when developing apps. These energy leaks are caused by inexperienced or new developers, also lack guideline and tools to detect energy inefficiency. In future work, we will work on a tool for detecting these energy leaks and estimate the energy consumption of the program. This tool will help developers to write energy-efficient apps.

7. ACKNOWLEDGEMENT

"This research was supported by the MISP(Ministry of Science, ICT), Korea, under the National Program for Excellence in SW)(2018-0-00192) supervised by the IITP(Institute of Information & communications Technology Planning & Evaluation)"(2018-0-00192).

REFERENCES

[1] Li, Li, et al. "Static analysis of android apps: A systematic literature review." *Information and Software Technology* 88 (2017): 67-95.

[2] Simon Hill, "Android vs. iOS: In-Depth Comparison of the Best Smartphone Platforms | Digital Trends," 2019. [Online]. Available: https://www.digitaltrends.com/mobile/android-vs-ios/. [Accessed: 31-Jul-2019].

[3] "Mobile Operating System Market Share Worldwide | StatCounter Global Stats." [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide. [Accessed: 10-

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Jun-2020].

[4] Liu, Kecheng, et al. "Security analysis of mobile device-to-device network applications." *IEEE Internet of Things Journal* 6.2 (2018): 2922-2932.

[5] M. V. J. Heikkinen, J. K. Nurminen, T. Smura, and H. Hämmäinen, "Energy efficiency of mobile handsets: Measuring user attitudes and behavior," *Telemat. Informatics*, vol. 29, no. 4, pp. 387–399, 2012.

[6] T. Das, M. Di Penta, and I. Malavolta, "A quantitative and qualitative investigation of performance-related commits in android apps," *Proc. - 2016 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2016*, pp. 443–447, 2017.

[7] L. Li, T. Riom, T. F. Bissyandé, H. Wang, J. Klein, and L. T. Yves, "Revisiting the impact of common libraries for android-related investigations," *J. Syst. Softw.*, vol. 154, pp. 157–175, 2019.

[8] R. Spolaor, E. D. Santo, and M. Conti, "DELTA: Data Extraction and Logging Tool for Android," *IEEE Trans. Mob. Comput.*, vol. 17, no. 6, pp. 1289–1302, 2018.

[9] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What Do Programmers Know about Software Energy Consumption?," *IEEE Softw.*, vol. 33, no. 3, pp. 83–89, 2016.

[10] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy API usage patterns in Android apps: an empirical study," pp. 2–11, 2014.

[11] Wu, Haowei, Shengqian Yang, and Atanas Rountev. "Static detection of energy defect patterns in Android applications." *Proceedings of the 25th International Conference on Compiler Construction*. 2016.

[12] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury, "EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 470–490, 2018.

[13] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, "Software-based energy profiling of Android apps: Simple, efficient and reliable?," *SANER 2017 - 24th IEEE Int. Conf. Softw. Anal. Evol. Reengineering*, pp. 103–114, 2017.

[14] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Inf. Softw. Technol.*, vol. 105, no. June 2018, pp. 43–55, 2019.

[15] Geiger, Franz-Xaver, and Ivano Malavolta. "Datasets of Android Applications: a Literature Review." *arXiv preprint arXiv:1809.10069* (2018).

[16] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?," *Proc. - 2015 30th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2015*, pp. 429–440, 2016.

[17] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," *Proc. - Int. Conf. Softw. Eng.*, vol. 1, pp. 89–99, 2015.

[18] H. Wu *et al.*, "Static window transition graphs for Android," *Autom. Softw. Eng.*, vol. 25, no. 4, pp. 833–873, 2018.

[19] D. Garbervetsky, E. Zoppi, and B. Livshits, "Toward full elasticity in distributed static analysis: The case of callgraph analysis," *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, vol. Part F1301, pp. 442–453, 2017.

[20] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of Android-specific code smells: The aDoctor project," *SANER 2017 - 24th IEEE Int. Conf. Softw. Anal. Evol. Reengineering*, pp. 487–491, 2017.

[21] Jabbarvand, Reyhaneh, et al. "Energy-aware test-suite minimization for android apps." *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016.

[22] Wu, Haowei. *Detection of Energy-Inefficiency Patterns in Android Applications*. Diss. The Ohio State University, 2018.

[23] Jenkins, John, and Haipeng Cai. "ICC-Inspect: supporting runtime inspection of Android inter-component

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

communications." *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 2018.

[24] Y. Zhao, M. S. Laser, Y. Lyu, and N. Medvidovic, "Leveraging Program Analysis to Reduce User-Perceived Latency in Mobile Applications," *Softarch.Usc.Edu*, 2018.

[25] Y. Liu, C. Xu, S. C. Cheung, and J. Lü, "GreenDroid: Automated diagnosis of energy inefficiency for Smartphone applications," *IEEE Trans. Softw. Eng.*, vol. 40, no. 9, pp. 911–940, 2014.

[26] J. Shim, K. Lim, S. Cho, S. Han, and M. Park, "Static and Dynamic Analysis of Android Malware and Goodware Written with Unity Framework," *Secur. Commun. Networks*, vol. 2018, pp. 1–12, 2018.

[27] C. H. P. Kim, D. Kroening, and M. Kwiatkowska, "Static program analysis for identifying energy bugs in graphics-intensive mobile apps," *Proc. - 2016 IEEE 24th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst. MASCOTS 2016*, pp. 115–124, 2016.

[28] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann, "EcoDroid: An Approach for Energy-Based Ranking of Android Apps," *Proc. - 4th Int. Work. Green Sustain. Software, GREENS 2015*, pp. 8–14, 2015.

[29] Fan, Lingling, et al. "Efficiently manifesting asynchronous programming errors in android apps." *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018.

[30] Jabbarvand, Reyhaneh, and Sam Malek. "μDroid: an energy-aware mutation testing framework for Android." *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017.

[31] Kim, Jeongmin, et al. "Enabling automatic protocol behavior analysis for android applications." *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 2016.

[32] L. Cruz and R. Abreu, "Performance-Based Guidelines for Energy Efficient Mobile Applications," *Proc. - 2017 IEEE/ACM 4th Int. Conf. Mob. Softw. Eng. Syst. MOBILESoft 2017*, pp. 46–57, 2017.

[33] Cruz, Luis, and Rui Abreu. "Using automatic refactoring to improve energy efficiency of android apps." *arXiv preprint arXiv:1803.05889* (2018).

[34] L. Wei, Y. Liu, and S.-C. Cheung, "OASIS: Prioritizing Static Analysis Warnings for Android Apps Based on App User Reviews," *ESEC/FSE '17 (Joint Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, vol. 11, pp. 672–682, 2017.

[35] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software," *IEEE Trans. Softw. Eng.*, vol. 43, no. 6, pp. 492–530, 2017.

[36] D. Octeau, D. Luchaup, S. Jha, and P. McDaniel, "Composite Constant Propagation and its Application to Android Program Analysis," *IEEE Trans. Softw. Eng.*, vol. 42, no. 11, pp. 999–1014, 2016.

[37] Y. Zheng, S. Kell, L. Bulej, H. Sun, and W. Binder, "Comprehensive multiplatform dynamic program analysis for Java and android," *IEEE Softw.*, vol. 33, no. 4, pp. 55–63, 2016.

[38] F. Toffalini, J. Sun, and M. Ochoa, "Practical static analysis of context leaks in Android applications," *Softw. - Pract. Exp.*, vol. 49, no. 2, pp. 233–251, 2019.

[39] S. Yang *et al.*, "Static window transition graphs for Android," *Autom. Softw. Eng.*, vol. 25, no. 4, pp. 833–873, 2018.

[40] Wu, Tianyong, et al. "Relda2: an effective static analysis tool for resource leak detection in Android apps." *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016.

[41] M. Hammad, H. Bagheri, and S. Malek, "DELDROID: An automated approach for determination and enforcement of least-privilege architecture in android," *J. Syst. Softw.*, vol. 149, pp. 83–100, 2019.

[42] Kuznetsov, Konstantin, et al. "Analyzing the user interface of Android apps." *Proceedings of the 5th International Conference on Mobile Software*

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Engineering and Systems. 2018.

[43] Torlak, Emina, and Satish Chandra. "Effective interprocedural resource leak detection." *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010.

[44] H. Huang, L. Wei, Y. Liu, and S.-C. Cheung, "Understanding and detecting callback compatibility issues for Android applications," *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng. - ASE 2018*, pp. 532–542, 2018.

[45] Borges, Nataniel P., Jenny Hotzkow, and Andreas Zeller. "DroidMate-2: a platform for Android test generation." *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018.

[46] Meng, Guozhu, et al. "Androvault: Constructing knowledge graph from millions of android apps for automated computing." *arXiv preprint arXiv: 1711.07451* (2017).

[47] M. Arnold, M. Vechev, and E. Yahav, "Qvm," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 1, pp. 1–35, 2011.

[48] A. Mukherjee and T. Chantem, "Energy management of applications with varying resource usage on smartphones," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2416–2427, 2018.

[49] L. L. Zhang, C. J. M. Liang, Y. Liu, and E. Chen, "Systematically testing background services of mobile apps," *ASE 2017 - Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pp. 4–15, 2017.

[50] Jabbarvand, Reyhaneh, Jun-Wei Lin, and Sam Malek. "Search-based energy testing of Android." *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019.

[51] A. Gupta, T. Zimmermann, C. Bird, N. Nagappan, T. Bhat, and S. Emran, "Mining energy traces to aid in software development," *Proc. 8th ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas. - ESEM '14*, pp. 1–8, 2014.

[52] Cruz, Luis, and Rui Abreu. "Catalog of energy patterns for mobile applications." *Empirical Software Engineering* 24.4 (2019): 2209-2235.

[53] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," *Proc. - Int. Conf. Softw. Eng.*, pp. 92–101, 2013.

[54] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," *Proc. 7th ACM Eur. Conf. Comput. Syst.*, pp. 29–42, 2012.

[55] Zhang, Lide, et al. "Accurate online power estimation and automatic battery behavior based power model generation for smartphones." *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 2010.

[56] Mittal, Radhika, Aman Kansal, and Ranveer Chandra. "Empowering developers to estimate app energy consumption." *Proceedings of the 18th annual international conference on Mobile computing and networking*. 2012.

[57] Liu, Yepang, et al. "DroidLeaks: a comprehensive database of resource leaks in Android apps." *Empirical Software Engineering* 24.6 (2019): 3435-3483.

[58] Wu, Haowei, Yan Wang, and Atanas Rountev. "Sentinel: Generating GUI tests for Android sensor leaks." *2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST)*. IEEE, 2018.

[59] Liu, Yepang, et al. "Understanding and detecting wake lock misuses for android applications." *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016.

[60] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "EARMO: An energy-aware refactoring approach for mobile apps," *IEEE Trans. Softw. Eng.*, vol. 44, no. 12, pp. 1176–1206, 2018.