CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, ISLAMABAD

A GUI Based Approach to Detect Energy Bugs in Android Applications

by

Ayesha Naseer

A thesis submitted in partial fulfillment for the degree of Master of Science

in the

Faculty of Computing
Department of Computer Science

2020

I dedicate my thesis to my husband, parents, supervisor and friends who encouraged and supported me. A special feeling of gratitude for my parents and husband for their endless love and support.

CERTIFICATE OF APPROVAL

A GUI Based Approach to Detect Energy Bugs in Android Applications

by

Ayesha Naseer

(MCS181012)

THESIS EXAMINING COMMITTEE

S. No.	Examiner	Name	Organization
(a)	External Examiner	Dr. Waseem Shahzad	FAST-NU, Islamabad
(b)	Internal Examiner	Dr. Nadeem Anjum	CUST, Islamabad
(c)	Supervisor	Dr. Aamer Nadeem	CUST, Islamabad

Dr. Aamer Nadeem
Thesis Supervisor
November, 2020

Dr. Nayyer Masood
Head
Dept. of Computer Science
November, 2020

Dr. Muhammad Abdul Qadir
Dean
Faculty of Computing
November, 2020

Author's Declaration

I, **Ayesha Naseer** hereby state that my MS thesis titled "**A GUI Based Approach to Detect Energy Bugs in Android Applications**" is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/abroad.

At any time if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my MS Degree.

(Ayesha Naseer)

Registration No:MCS181012

Plagiarism Undertaking

I solemnly declare that research work presented in this thesis titled "**A GUI Based Approach to Detect Energy Bugs in Android Applications**" is solely my research work with no significant contribution from any other person. Small contribution/help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of MS Degree, the University reserves the right to withdraw/revoke my MS degree and that HEC and the University have the right to publish my name on the HEC/University website on which names of students are placed who submitted plagiarized work.

(Ayesha Naseer)
Registration No:MCS181012

Acknowledgements

I am obliged to Allah Almighty the Merciful, the Beneficent and the source of all Knowledge, for granting me the courage and knowledge to complete this thesis. He bestowed me with the patience, wellness and understanding to complete my thesis.

I am also thankful to my husband and parents for their love support and sacrifices which they have made for me. Especially my husband who played an important role in guiding me and keeping me motivated during the thesis.

A special thanks to **Dr. Aamer Nadeem** for his guidance, motivation and regular feedback during my thesis. I am sincerely grateful to him for his continuous support during my research. He is the most respectable man and working under his supervision was an honor for me.

I also want to thanks Mr. Qamar Zaman and Asia Shahab for their support and motivation. I am truly grateful how they keep me motivated during my research. I would also like to thank the CSD research group for their guidance. Finally, I would like to thank everyone who helped and supported me.

(Ayesha Naseer)

Registration No:MCS181012

Abstract

Smartphones have improved in terms of their processing speed and memory capacity. The increased usage of smartphones has led to the widespread usage of smartphone applications. If the applications are not carefully developed, they become energy-inefficient due to some energy-intensive resources used in them (such as Wi-Fi, GPS, etc.). When these resources are left unreleased after use they start consuming the smartphone's battery power even if no application is using these resources. This behavior is called resource-leaking and causes energy-bugs.

Different techniques exist in the literature that can detect the presence of energy-bugs due to resource-leaking. They can be classified into static, dynamic, and hybrid techniques. Static techniques do not execute the code therefore they may include infeasible paths and generate false-positives. Dynamic techniques can detect the bugs at run-time by executing the code which static analysis techniques cannot detect. However, they become costly because multiple paths have to be executed. Hybrid techniques provide the benefit of both static and dynamic analysis techniques.

The existing techniques detect the energy-bugs at the method level. A method containing the code of an event-handler corresponding to an event. They consider each method in isolation and check if the particular method has acquired a resource in it but did not release that resource within that method rather it is released in another method than they call it an energy-bug at the method level. However, if we consider both the methods at the application level, where both the methods execute in a single path then the resource gets released and there is no energy-bug which shows that the existing techniques generate false-positives.

In this thesis, we enhance an existing technique by performing energy-bug detection at the application level and propose an approach that does not produce false-positives. In the test-paths of the Event Flow Graph generated at the application level, when we execute two such methods in which a resource is acquired in one of them and released in the other, then, in this case, it is not an energy-bug because a resource is being released after the acquisition and there is no false-positive at the application level. We have evaluated our approach with 11 real-life Android applications from different online sources and detected energy-bugs in them. We have compared the proposed approach with the existing approach and found false-positives in their approach.

Contents

List of Figures

List of Tables

Abbreviations

APK	Application Package Kit
CFG	Control Flow Graph
EFG	Event Flow Graph
FNs	False Negatives
FPs	False Positives
TP	Test Path
TR	Test Requirement

Chapter 1

Introduction

In the recent era of technology, smartphones have improved in terms of their processing speed and memory capacity. They come up with a wide range of I/O components and sensors, such as Wi-Fi, GPS, and so on [1]. Due to these components, application developers are developing complex applications for smartphones, which are often energy-inefficient [2]. These energy-inefficiencies limit the battery power of the smartphone. We can categorize the energy-inefficiencies in the smartphone applications into energy-bugs and energy-hotspots [2]. We say that an application is energy-inefficient due to an energy-bug when it prevents the smartphone from becoming idle even after it has been closed/completed its execution and there is no user activity [2]. Whereas an application is energy-inefficient due to energy-hotspot is a scenario, in which an application is executing on a smartphone and it starts consuming a high amount of battery power [2]. An energy-bug arises when an application acquires a resource in the code according to its need but the developer might omit the call to release a resource after acquiring it. This causes resource-leaking which leads to energy-consumption and causes energy-bug.

We focus on the detection of energy-bugs in Android applications due to resource leaks i.e. such resources that should be manually released by the developers but they failed to do so [1] and as a result, such buggy applications start consuming the smartphone battery.

1.1 Classification of Energy-Bugs

The energy-bugs that can be present in an Android application can be classified as:

- **Resource Leak:** An application must release all the resources (such as Wi-Fi, GPS) it acquires during execution [2] [3].
- **Wakelock Bug:** In Android, a power management mechanism that helps an application to indicate that the device needs to stay awake. If we inappropriately use the Wakelocks, this can cause the device to be stuck in a high-power state even if the application has finished its execution. We call this situation a Wakelock bug [2] [3].
- **Vacuous Background Services:** Background services are expensive in terms of energy consumption. A vacuous background service bug occurs when an application mishandles such services [2] [3].
- **Immortality Bug:** If we close an application while it is still in high power-consumption state then it creates an immortality bug [2] [3].

1.2 Resource Acquisition in Android Applications

There are two ways to acquire a resource in an application. First, is a local resource that can be declared in a function and we assume that the developer releases it at the end of the function. Second, is a global resource that can be declared outside all the functions and within a class and that must be released on all exit-paths from its request point. We should focus on all exit-paths to detect missing resource operations to detect resource leaks [4]. The applications containing an energy-bug may not fail or stop during execution due to which it is difficult to locate energy-bugs in them. Besides, the developers only focus on the functionality of the application; they do not conduct performance testing before releasing the application [1].

1.3 Energy-Bug Detection Techniques

The existing techniques perform energy-bug detection at the method level by performing static or dynamic analysis technique.

A method that represents an event event-handler corresponding to an event within the application.

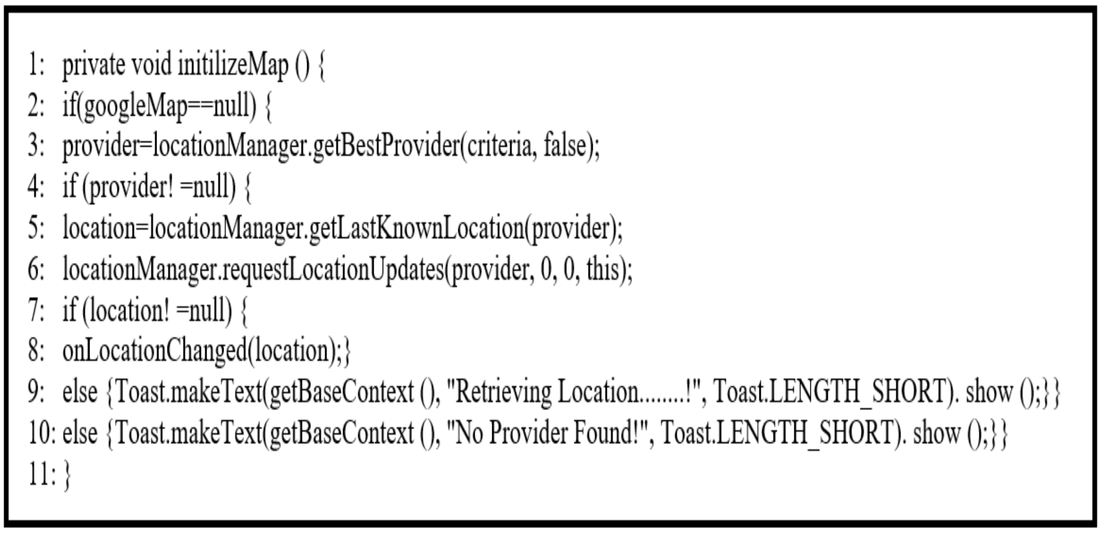
They generate a Control Flow Graph of each method and when a resource is acquired at any line of a method and is not released by the developer, they call it an energy-bug.

Figure 1.1 shows that a location-update resource is acquired at line 6 of the method but the developer did not release the resource which causes an energy-bug.

While at the application level, if a resource is acquired in one method and released in the other method, and both the methods execute in the paths of the Event Flow Graph it becomes a false-positive.

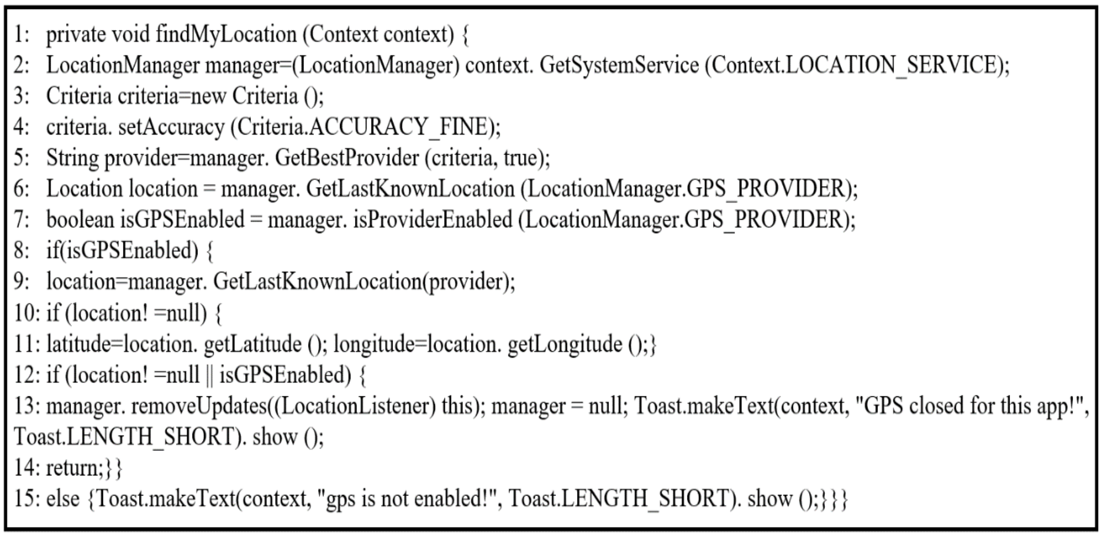
An example code of a method is shown in Figure 1.1 in which a location-update resource is acquired in line 6.

Figure 1.2 shows the example code of a method in which the location-update resource acquired in Figure 1.1 is released in line 13.



```
1:  private void initilizeMap () {
2:  if(googleMap==null) {
3:  provider=locationManager.getBestProvider(criteria, false);
4:  if (provider! =null) {
5:  location=locationManager.getLastKnownLocation(provider);
6:  locationManager.requestLocationUpdates(provider, 0, 0, this);
7:  if (location! =null) {
8:  onLocationChanged(location);}
9:  else {Toast.makeText(getBaseContext (), "Retrieving Location........!", Toast.LENGTH_SHORT). show ();}}
10: else {Toast.makeText(getBaseContext (), "No Provider Found!", Toast.LENGTH_SHORT). show ();}}
11: }
```

Figure 1.1: An Example Code of Method Acquiring a Resource



```
private void findMyLocation (Context context) {
LocationManager manager=(LocationManager) context. GetSystemService (Context.LOCATION_SERVICE);
Criteria criteria=new Criteria ();
criteria. setAccuracy (Criteria.ACCURACY_FINE);
String provider=manager. GetBestProvider (criteria, true);
Location location = manager. GetLastKnownLocation (LocationManager.GPS_PROVIDER);
boolean isGPSEnabled = manager. isProviderEnabled (LocationManager.GPS_PROVIDER);
if(isGPSEnabled) {
location=manager. GetLastKnownLocation(provider);
if (location! =null) {
latitude=location. getLatitude (); longitude=location. getLongitude ();}
if (location! =null || isGPSEnabled) {
manager. removeUpdates((LocationListener) this); manager = null; Toast.makeText(context, "GPS closed for this app!", Toast.LENGTH_SHORT). show ();
return;}}
else {Toast.makeText(context, "gps is not enabled!", Toast.LENGTH_SHORT). show ();}}}
```

FIGURE 1.2: An Example Code of Method Releasing a Resource

When we execute the above two methods in the paths of the Event Flow Graph, it becomes a false-positive. But if we execute these event-handlers in isolation, they cause an energy-bug. Several static and dynamic analysis techniques exist in the literature for the detection of energy-bugs in Android applications.

1.3.1 Static Analysis Techniques

Static analysis techniques work in a non-run-time environment and detect any possible bug in the code while the code is not executing at all. Static analysis techniques take longer to detect bugs from a large-scale application [5]. These techniques cannot detect those bugs which can be detected by the dynamic analysis techniques at run-time and therefore generate false-positives [6].

1.3.2 Dynamic Analysis Techniques

Dynamic analysis techniques work in a run-time environment while the code is executing and can detect the bugs which cannot be detected by the static analysis techniques [6] and remove false-positives. However, dynamic analysis techniques become costly because multiple paths have to be executed. Therefore, dynamic analysis techniques are better than static analysis techniques.

1.3.3 Hybrid Analysis Techniques

Hybrid approaches also exist in the literature which is the combination of both static and dynamic analysis techniques [2] [3]. However, they generate false-positives due to static analysis and are costly due to dynamic analysis.

We have chosen a hybrid approach as a base that detects the energy-bugs statically at the application level [3]. They generate an Event Flow Graph of the application. The nodes of the Event Flow Graph represent the events and the edges represent the sequence of user-interactions or event-sequences in which the events execute [2]. The paths of the Event Flow Graph represent the flow of event-sequences in which they execute. They associate each node of the Event Flow Graph with its corresponding event-handler and generate a Control Flow Graph of each event-handler. They focus on those events in the paths of the Event Flow Graph whose corresponding event-handler has acquired a resource and find the exit event-node whose corresponding event-handler has released that particular resource. Otherwise, this becomes an energy-bug. They statically generate the event-sequences (paths) of the Event Flow graph and do not apply any graph coverage criterion to generate the paths due to which some paths might be missed and it results in false-positives.

1.4 Problem Statement

The existing techniques perform energy-bug detection at the event-handler level. When a developer acquires a resource in an event-handler and omits a call to release that resource in that particular event-handler, it becomes an energy-bug at the event-handler level. At the application level when a resource is acquired in an event-handler and released in another event-handler, and when both the event-handlers execute in the path of the Event Flow Graph at the application level, it makes that energy-bug detected at the event-handler level a false-positive because the acquired resource is released at the application level. The existing techniques do not focus on application level bug detection. They focus on the event-handlers in isolation and therefore generate false-positives.

1.5 Research Questions

For this research, the following factors must be taken into account:

RQ 1: What are the gaps in the existing techniques?

We have carried out the literature survey to find the gaps in the existing techniques for the detection of energy-bugs in Android applications.

RQ 2: How to enhance the existing techniques for application level energy-bug detection?

We have proposed an approach that enhances an existing technique for the application level energy-bug detection and removes false-positives.

RQ 3: Is the proposed technique better in terms of eliminating false-positives?

We have performed experiments on different real-life Android applications for the comparison of existing techniques with the proposed technique.

We have eliminated false-positives generated in the existing technique.

Our research is carried out to answer the above-mentioned research questions.

1.6 Research Methodology

1. First of all, we have done the literature review to find the common techniques that are relevant to the detection of energy-bugs in Android applications.

   After studying the various techniques, we conclude that the existing techniques detect the energy-bugs at the event-handler level only.

They do not consider the application level bug detection and generate false-positives.

2. To overcome the gap in exiting techniques, we have enhanced the existing technique for the detection of energy-bugs at the application level by identifying that the energy-bugs detected at the event-handler level are false-positives at the application level.

3. The implementation of our proposed technique is performed in the following phases:

   (a) In the first phase, we generate the Event Flow Graph of each Android application. Each node of the Event Flow Graph represents an event of the Application and the edges represent the sequence of user-interactions.

   (b) In the second phase, we generate the test-path of the Event Flow Graph by applying the edge-coverage criterion.

       The test-paths generated from the Event Flow Graph represent the events that occur in the Event Flow Graph.

   (c) In the third phase, we generate the Control Flow Graph of those events from the test-paths of the Event Flow Graph whose corresponding event-handler acquires or releases a resource.

       We associate the Control Flow Graph of each event-handler with its corresponding event in the Event Flow Graph.

       After that, we generate test-paths of each Control Flow Graph.

       We replace these test-paths with their corresponding events that occur in the test-paths of the Event Flow Graph to make a complete test-path.

   (d) In the last phase, we make test inputs for each complete test-path and evaluate them for the presence of energy-bugs.

4. We compare our proposed technique with the existing technique and verify that the energy-bugs detected from the existing technique at the event-handler level are false-positives.

1.7 Research Contribution

In this research work, we have enhanced an existing technique that will identify the energy-bugs detected at the event-handler level by making a Control Flow Graph are false-positives at the application level. We generate the Event Flow Graph for an application and generate its test-paths by applying the graph coverage criterion. We also generate the Control Flow Graph for each event that occurs in the test-paths of the Event Flow Graph and generate its test-paths by again applying the graph coverage criterion. We make a complete test-path by using the test-paths of both the Control Flow Graph and the Event Flow Graph. In this way, we get multiple complete test-paths against each test-path of the Event Flow Graph. We evaluate these complete test-paths and identify that the energy-bugs detected at the event-handler level are false-positives at the application level.

1.8 Thesis Structure

For the sake of clarity and understanding, the thesis is divided into the following chapters:

- Chapter 1 describes the introduction of the proposed technique and its objective.
- Chapter 2 describes the literature survey in which we researched the existing techniques.
- Chapter 3 describes the proposed solution and implementation of the proposed solution.
- Chapter 4 describes the experimental results of different Android applications.
- Chapter 5 describes the research questions and the conclusion.

Chapter 2

Literature Review

This chapter describes several static, dynamic and hybrid approaches that exist in the literature for the detection of energy-bugs in Android applications due to resource-leaking. It also describes the comparison between the existing techniques.

2.1 Static Approach

The static approach performs the static analysis of the code to check the defects in the code without executing the code of the application. It identifies the possible bugs in the code but generates false-positives. Static approaches take longer to detect bugs from a large-scale application [5].

2.2 Dynamic Approach

The dynamic approach analyzes the dynamic behavior of the code by executing the code. The dynamic approach can always find the errors at the run-time that a static approach cannot find [6]. Therefore, it is better than the static approach. However, they become costly because multiple paths have to be executed. The dynamic approach executes the application; therefore, chances are higher to find bugs in the application than a static approach. That is why we have not considered static approaches in the literature study.

2.3 Hybrid Approach

The hybrid approach is a combination of both static and dynamic approaches. It provides the benefit of both static and dynamic approaches [2] [3]. It generates the false-positives due to static approach and is costly due to dynamic approach.

2.4 Literature Studies

In 2012, Zhang et al. [7] presented an automated detector of energy leaks called ADEL. It tracks the information flow of network traffic within an application. It helps to identify resource- leaks due to network communications (such as Wi-Fi and 3G). It uses dynamic taint tracking to detect energy leaks and labels each data object with a tag and tracks the network data from its origination to its use or until its deletion. They evaluated their approach with 15 real-world Android applications and found energy leaks in 6 of them. The common causes of energy-leaks were the inefficient data refreshing behavior and APIs were misinterpreted. They neglected control-flow and only focused on data-flow due to which false-positives were generated.

In 2013, Liu et al. [4] presented an automated approach, which focuses on sensory data utilization at different states and locates energy inefficiency problems. It implements this approach as a tool named GreenDroid. It derives the Application Execution Model (AEM) from specifications that simulate an application's runtime behavior and specifies calling relationships between event handlers. It checks which sensor listeners have forgotten to unregister at the end of the execution. They evaluated their approach with 6 open-source Android applications. They analyzed each Android application for its sensory data utilization. The common causes of energy-leaks were due to sensory data utilization and sensor-listener misusage. Complex test-inputs could not be generated in their approach due to which false-negatives are generated.

In 2014, Liu et al. [8] presented a dynamic approach to detect the energy-inefficiency problems in the Android applications due to missing deactivation of sensors or wake-locks and cost-ineffective use of sensor data. They have enhanced the approach proposed in [4] by exploring the state space of an application. They monitor its sensory data utilization and usage of sensors and wakelocks. It generates detailed reports to help developers locating energy problems in the Android applications. They evaluated their approach with 13 open-source Android applications. They found energy-related problems in 11 of them. The common cause of energy-leaks was due to missing wakelock deactivation, missing sensor deactivation, and sensory data was underutilized. Their proposed approach was unable to generate complex inputs such as gestures or video inputs and generated false-negatives.

In 2014, Banerjee et al. [2] presented an automated test generation framework for the detection of energy hotspots/bugs in Android applications. The framework consists of two components: the guidance component and the hotspot/bug detection component. In the guidance component, they discuss the exploration of event traces to reveal energy hotspots/bugs. In the hotspot/bug detection component, they execute an application on a smartphone and attach a power meter with it. For a given event trace, the power meter measures the power consumption of the application. They evaluated their approach with 30 Android applications that are freely available and reported energy-bugs in 10 them. Their proposed approach was unable to reach all the states of the GUI of an Android application. Their proposed approach generated false-positives because some portions of code could not be analyzed.

In 2015, Abbasi et al. [9] presented an operational definition of resource-leak problems. They described the scenario with the help of a framework for the presence of energy bugs. They have detected the energy-bugs when the applications or the platforms update. They analyzed that the detection of energy-bugs in Android applications is difficult to perform because the applications do not stop/fail if an energy-bug is present in them. Their focus was on resource-leak problems. However, they did not implement the framework to detect energy bugs. They evaluated their approach using a browser application and the YouTube application. They did not implement their approach to detect the energy-bugs from the Android applications.

In 2016, Wang et al. [10] presented an extension for GreenDroid [8] that provides the ability to support new features of Android 5.0. It focuses on sensor-listener misusage and wakelock misusage. It provides a state machine based on Application Execution Model (AEM). They have also better organized the reports represent the important information. However, it neglected some patterns of wakelock misuses like multiple wakelock acquisitions. They evaluated their approach by using 13 real-world Android applications. Their results proved that the extension they have done for the GreenDroid named E:GreenDroid was effective. However, they missed some patterns of wakelock misuses and generated false-negatives. They were unable to detect multiple wakelock acquisition [10].

In 2017, Li et al. [11] presented another extension for GreenDroid [8]. They developed a tool named CyanDroid. It focuses on generating sensory data; it tracks the propagation of sensory data and analyses its utilization. They also analyze whether the sensory data is being used cost-effectively. They systematically generate the sensory data and enhance the GreenDroid for the detection of energy-bugs in Android applications. They evaluated their approach by using 4 real-world Android applications and reported energy bugs in all of them. They were unable to simulate the complex user-events and generated false-negatives.

In 2017, Liu et al. [12] presented an approach to extend GreenDroid [7] and developed a tool named NavyDroid. It focuses on multiple patterns of wake-locks misuses. They proposed an approach that consists of a monitor and simulation part. In the simulation; they explore the state space of an application, and in the monitor part; they monitor the presence of energy-bugs. They evaluated their approach by using 17 real-world Android applications. They evaluated more energy-bugs problems than the E:GreenDroid [10]. However, they were unable to detect multiple wakelock acquisition and generated false-negatives.

In 2018, Banerjee et al. [3] presented both static and dynamic approaches for the detection of energy bugs in an application. In [2], they generated the Event Flow Graph of the user-generated events. Whereas, in this approach, they have generated Control Flow Graph of the event-handlers along with each event of the Event Flow Graph. They associate each event of an Event Flow Graph with the Control Flow Graph of the corresponding event-handler. They evaluated their approach by using 35 Android applications from online sources and reported energy-bugs in 12 of them. 8 of the applications had energy-bugs due to GPS. 2 of the applications had energy-bugs due to GPS and sensors and 2 of them had energy-bugs due to improper wakelock usage. Their approach statically analyzes the code and generated overestimated results that also include infeasible paths and generates false-positives [3].

In 2018, Abbasi et al. [13] presented an approach that focuses on tail energy bugs and identified their root causes. They developed a Java-based tool to detect the presence of Application Tail Energy Bugs (ATEBs) and check the behavior of activities or services of the Android applications in the presence of wakelocks. They measure the power-consumption reading of each application. They evaluated their approach with the help of 32 experiments and observed the behavior of application's components in the presence or absence of wakelocks. They did not provide any information regarding how to release wakelocks automatically once they have been acquired.

In 2020, Li et al. [48] presented an approach that focuses on the detection of energy-bugs due to resource leaks such as Wi-Fi, GPS. They evaluated their approach by using 27 real-life Android apps like Chrome and Firefox.

The main root cause detected for energy-inefficiency was unnecessary workload. However, their approach eliminated false positives as well.

2.5 Analysis and Comparison

Table 2.1 lists the comparison of all the techniques proposed in the literature. Several energy problems are discussed in the literature related to network communications (such as Wi-Fi and 3G), missing deactivation of sensors/wake-locks, hardware components, etc. Some of them follow the white box testing strategy to generate test cases and some of them follow the black box testing strategy. Some of them follow data flow analysis to detect energy bugs that also includes infeasible paths and generate false positives. Some use Application Execution Model (AEM) to simulate the runtime behavior of an application. The table 2.1 represents the year in which each literature study is published, the technique used in each literature study, objective of each literature study and the weaknesses in them.

TABLE 2.1: Literature Review

Ref. No.	**Sr. No.**	**Year**	**Technique used**	**Objective**	**Weaknesses**
[7]	1.	2012	Dynamic	Focus on detection of energy bugs due to resource leaks i.e. unnecessary network communication	•Tracks data-flow, neglects control-flow & reports FNs •Performs method-level bug detection & reports FPs
[4]	2.	2013	Dynamic	Focus on detection of energy-inefficiency problems due to resource leaks i.e. sensory data utilization	•Cannot generate complex inputs & reports FNs •Performs method-level bug detection & reports FPs

Ref. No.	**Sr. No.**	**Year**	**Technique used**	**Objective**	**Weaknesses**
[8]	3.	2014	Dynamic	Focus on detection of energy problems due to wakelocks and resource leaks i.e. sensors	•Cannot generate gesture and video inputs & reports FPs •Performs method-level bug detection & reports FNs
[2]	4.	2014	Hybrid	Focus on detection of energy- hotspots and energy-bugs	•Performed app-level but detection bug neglected coverage criterion & reports FNs
[9]	5.	2015	Dynamic	Focus on detection of energy-bugs due to resource leaks i.e. Wi-Fi, GPS	•Do not consider upgraded features and functionality of app & reports FNs •Performs method-level bug detection & report FPs
[10]	6.	2016	Dynamic	Focus on detection of energy problems due to wakelocks misuse and resource leaks i.e. sensors	•Neglects some patterns of wakelocks & reports FNs •Performs method-level bug detection & reports FPs

Ref. No.	**Sr. No.**	**Year**	**Technique used**	**Objective**	**Weaknesses**
[11]	7.	2017	Dynamic	Focus on detection of energy-inefficiency problems due to resource leaks i.e. sensory data utilization	•Cannot simulate complex inputs & reports FNs •Performs method-level bug detection & reports FPs
[12]	8.	2017	Dynamic	Focus on detection of energy problems due to wakelocks and resource leaks i.e. sensors	•Cannot address unnecessary patterns of wakelock misuses & reports FNs •Performs method-level bug detection & reports FPs
[3]	9.	2017	Hybrid	Focus on detection of energy-bugs due to wakelocks and resource leaks i.e. GPS, Sensors	•Performs app-level bug detection, neglects coverage criterion & reports FNs •Does not generate all complete test-paths & reports FPs

Ref. No.	**Sr. No.**	**Year**	**Technique used**	**Objective**	**Weaknesses**
[13]	10.	2018	Dynamic	Focus on detection of energy bugs due to wakelocks and resource leaks i.e. GPS	•App components like audio and wireless services are not considered & reports FNs •Performs method-level bug detection & reports false positives
[48]	11.	2020	Hybrid	Focuses on detection of energy issues due to resource leaks i.e. Wi-Fi, GPS	•Includes infeasible paths & reports FPs •Performed method-level bug detection & reports false positives

It is concluded that the existing literature studies detect different types of energy bugs. After analyzing the existing studies, we conclude that the existing proposed solutions consider both static and dynamic analysis techniques for detecting energy bugs but we focused on dynamic and hybrid techniques. Besides that, existing studies only focus on the detection of energy-bugs at the method level. They do not focus on application-level energy-bug detection and therefore generate false-positives.

Chapter 3

Proposed Approach

In the literature survey, we have identified the most relevant existing studies related to energy bugs detection in Android applications. Some of the studies are based on dynamic detection of energy-bugs and few of them are hybrid approaches. Existing studies do not focus on the detection of energy bugs at the application-level. They only focus on the method level energy bug detection.

When energy-bugs are detected at the method level using Control Flow Graph (CFG), it is considered that if a resource is acquired at any line in the method then it must be released within that particular method where the resource has been acquired.

We have chosen a hybrid approach [3] to extend my work. Hybrid approaches provide the benefit of both static and dynamic approaches.

However, the hybrid approach that we have chosen to extend is using a static analysis technique for the detection of energy bugs.

They have worked on application-level by making the Event Flow Graph of the application. They associate each event with its corresponding event-handler and detect the energy bugs in those event-handlers. They do not detect energy-bugs at the application level and therefore, generating false-positives. Furthermore, they have not used any coverage criterion in their approach and that is why their approach is reporting false-negatives as well. To overcome the gap in the existing studies, we have enhanced an existing technique [3].

Our proposed approach aims to enhance the existing technique by detecting false-positives from the existing technique [3] and shows the energy-bugs detected at the method level using Control Flow Graph are not bugs at the application-level but are false-positives.

Our proposed approach detects only real energy-bugs in Android applications and eliminates false-positives.

3.1 Proposed Solution

The flowchart of our proposed solution is shown in Figure 3.1.

We take APK file of each application as an input and generate the Event Flow Graph of each application.

The Event Flow Graph is generated by using the Monkey tool. It is a command-line tool that sends a stream of random user-events to the system and generates Event Flow Graph.

Once the Event Flow Graph is generated, we make test-paths of the graph by applying the edge-coverage criterion.

It is a graph-coverage criterion that covers all the edges of the graph.

In parallel, we take source code of each application and generates the Control Flow Graph of the events in which a resource s acquired or released and make test-paths of the Control Flow Graph by also applying the criterian named as edge-coverage criterion.

Then, we combine test-paths of both the Event Flow Graph and the Control Flow Graph and make complete test-paths. We make test-inputs for each complete test-path and verify the existence of energy-bugs in the applications.

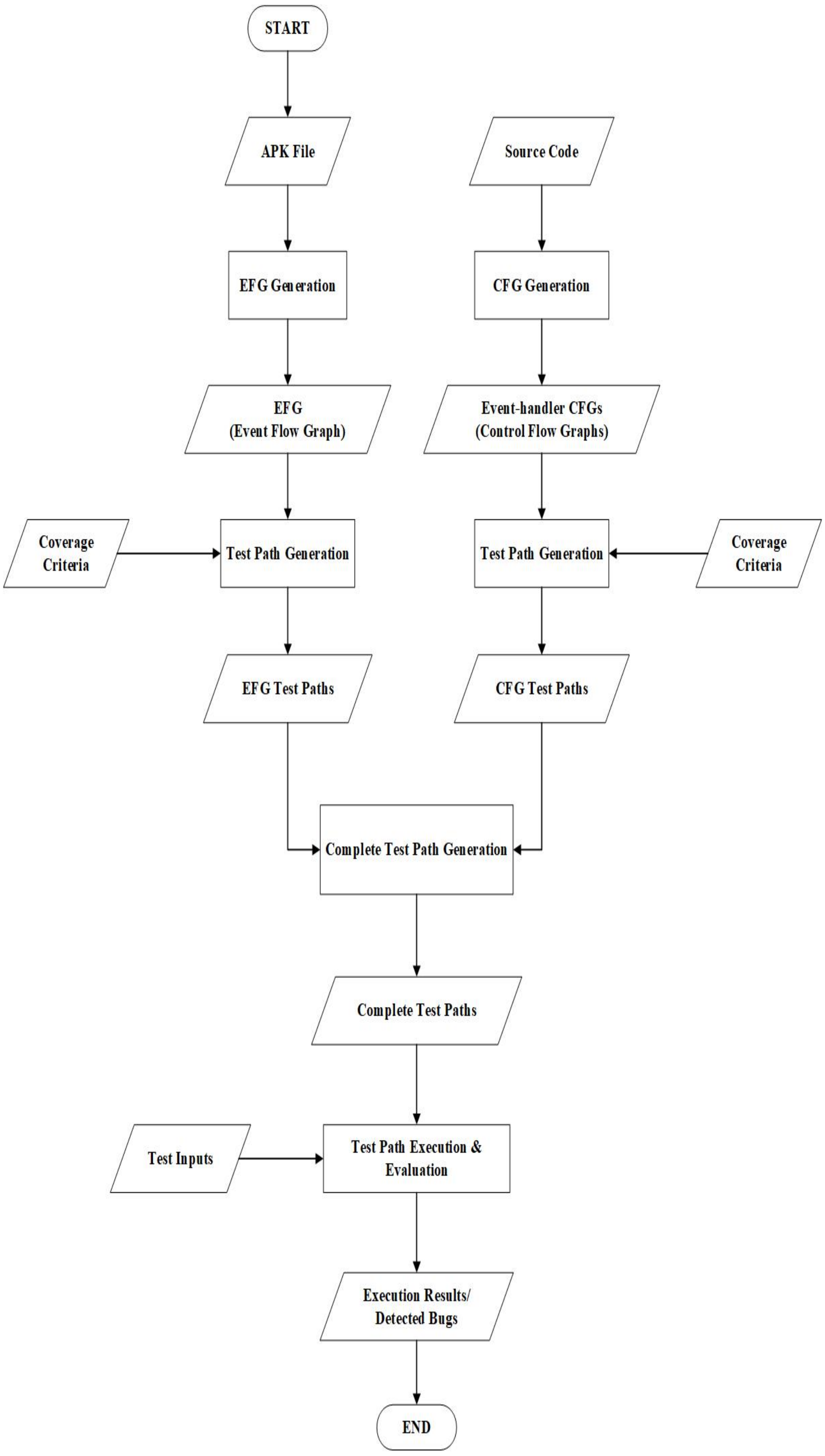


FIGURE 3.1: Flowchart of Proposed Solution

An example application is used to describe the processes of our proposed solution in detail.

This example is based on location tracking by using GPS.

A friend can track the location of another friend and they both can share their locations with each other.

This example contains 8 event-handlers in it against each event.

3.1.1 Event Flow Graph Generation

The Event Flow Graph is abbreviated as EFG. It is a graph representation of Graphical User Interface (GUI) whose nodes represent user-events and the edges represent a sequence of user-interactions i.e.an edge from event e1 to event e2 represents that the event E2 can be performed immediately after the event E1 [14].

The Event Flow Graph is generated for the whole application.

An application contains events and the corresponding event-handlers to each event.

Each event becomes a node of the Event Flow Graph. The Event Flow Graph is generated using the Monkey tool for each particular application.

A monkey is a command-line tool that can be run on an emulator.

It takes the apk file as an input and generates a stream of random user-events such as touch events, click events, or gesture events [15] for a particular application.

These events represent the flow in which they are executed in an application and helps in creating the Event Flow Graph for each application.

The Event Flow Graph as a whole represents the flow in which the events are executed within an application.

Figure 3.2 represents the Event Flow Graph of the example application.

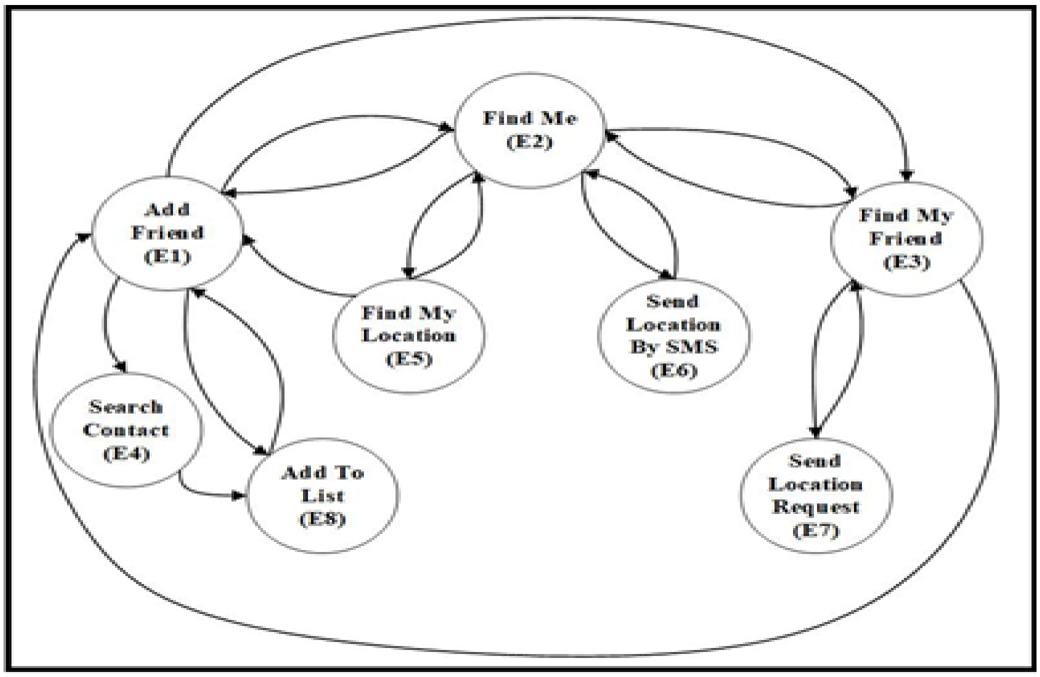
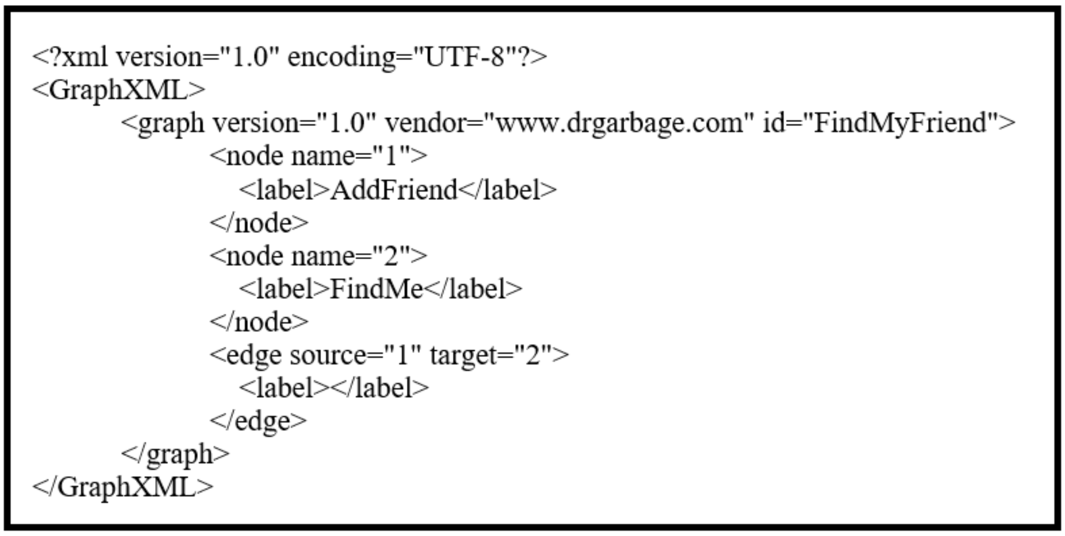


FIGURE 3.2: Event Flow Graph of the Example Application

3.1.2 Test-Path Generation for the Event Flow Graph

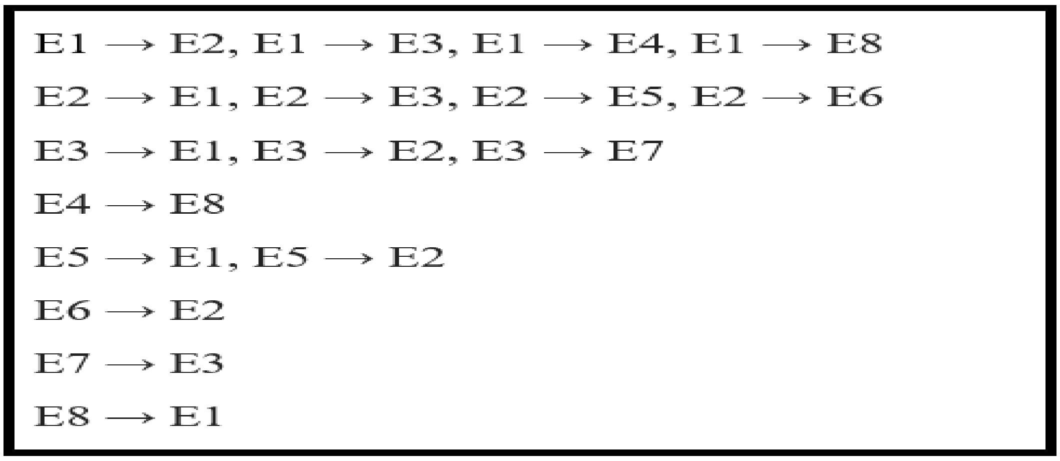
A test-path is a complete path that starts from the initial node of the graph and ends at its final node. In our case, a test-path is a sequence of events and contains multiple events in it. It is an execution path within the whole application. We convert the Event Flow Graph into XML code. The sample XML code of the Event Flow Graph shown in Figure 3.2 is given below in Figure 3.3:



```
<?xml version="1.0" encoding="UTF-8"?>
<GraphXML>
        <graph version="1.0" vendor="www.drgarbage.com" id="FindMyFriend">
                <node name="1">
                   <label>AddFriend</label>
                </node>
                <node name="2">
                   <label>FindMe</label>
                </node>
                <edge source="1" target="2">
                   <label></label>
                </edge>
        </graph>
</GraphXML>
```

FIGURE 3.3: Sample XML Code of the Event Flow Graph shown in Figure 3.2

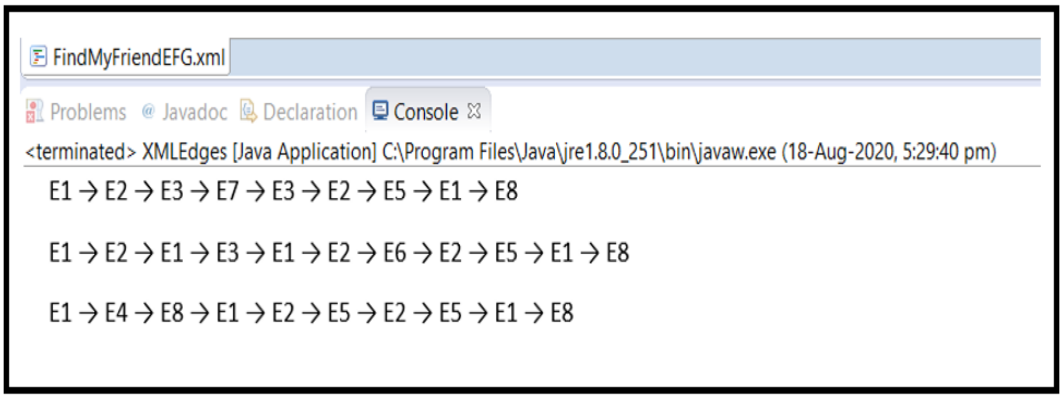
The node name in the XML code represents the event number and label shows the name of the event. Source node represents the event number (E1) from where the edge is going to a target node which also represents the event number (E2). The edge from source node (event E1) to target node (event E2) represents that the event E2 can be performed immediately after the event E1. For generating the test-paths of the Event Flow Graph, we pass the XML file as an input to a program and apply the edge coverage criterion on it. The edge coverage criterion is a graph coverage criterion. The test requirement for any coverage criterion is that it should cover all the requirements. Therefore, the test requirement for edge coverage criterion is that it should cover all the edges of the graph. The edges of the Event Flow Graph shown in Figure 3.2 are shown below in Figure 3.4:



E1 ⟶ E2, E1 ⟶ E3, E1 ⟶ E4, E1 ⟶ E8
E2 ⟶ E1, E2 ⟶ E3, E2 ⟶ E5, E2 ⟶ E6
E3 ⟶ E1, E3 ⟶ E2, E3 ⟶ E7
E4 ⟶ E8
E5 ⟶ E1, E5 ⟶ E2
E6 ⟶ E2
E7 ⟶ E3
E8 ⟶ E1

FIGURE 3.4: Edges of the Event Flow Graph for Aripuca Application

After applying the edge coverage criterion, we get the test-paths of the Event Flow Graph as an output. The output is shown in Figure 3.5.



```
<terminated> XMLEdges [Java Application] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (18-Aug-2020, 5:29:40 pm)
  E1 → E2 → E3 → E7 → E3 → E2 → E5 → E1 → E8

  E1 → E2 → E1 → E3 → E1 → E2 → E6 → E2 → E5 → E1 → E8

  E1 → E4 → E8 → E1 → E2 → E5 → E2 → E5 → E1 → E8
```

FIGURE 3.5: Test-Paths of the Event Flow Graph shown in Figure 3.2

We have made sure that all the edges of the graph are covered by the generated test-paths.

The edges covered from test-path 1 are:

E1$\rightarrow E2, E1 \rightarrow E8, E2 \rightarrow E3, E2 \rightarrow E5, E3 \rightarrow E2, E3 \rightarrow E7, E5 \rightarrow E1, E7 \rightarrow E3$

The edges covered from test-path 2 are:

E1$\rightarrow E3, E2 \rightarrow E1, E2 \rightarrow E6, E3 \rightarrow E1, E6 \rightarrow E2$

The edges covered from test-path 3 are:

E1$\rightarrow E4, E4 \rightarrow E8, E5 \rightarrow E2, E8 \rightarrow E1$

Hence, all the edges of the Event Flow Graph are covered by the test-paths.

3.1.3 Complete Test-Path Generation

Complete test-paths are those paths that are generated after combining test-paths of both the CFG and the Event Flow Graph. CFG is generated for the event-handlers. We have generated the CFG by using the CFG Factory plugin available in Eclipse software. This plugin generates the Control Flow Graph by using the Java code and exports it in the XML format, DOT format, and several image formats. CFG is a graph whose nodes represent a statement number of the program and the edges represent the flow in which these statements are executed [16]. When the test-paths are generated for the EFG, we choose those events from each test-path whose corresponding event-handler acquires or releases a resource and generate a Control Flow Graph for each of these event-handlers. We associate each CFG with its corresponding event. Figure 3.6 represents the CFG associated with the events of the EFG. It is to be noted that after analyzing the source code of this example, we have mentioned only the CFG of those event-handlers in which the resource is being acquired or released. Other event-handlers in which a resource is neither acquired nor released are not mentioned. The events E2, E5, and E6 are the events from each test-path of the Event Flow Graph whose corresponding event-handlers have acquired or released a resource in their code. The Control Flow Graph of the other event-handlers is not generated because no resource is acquired or released in them and therefore, we cannot find any energy-bug in their path. After generating the Control Flow Graph for the event-handlers, we export the Control Flow Graph in the XML file format. We have generated the Control Flow Graph by using the Control Flow Graph Factory plugin available in the Eclipse software. With the help of this plugin, we can convert the Control Flow Graph in different file formats such as XML file format, DOT file format. We pass this XML file as an input to the program. The program applies the edge-coverage criterion on it and generates the test-paths for the Control Flow Graph. These test-paths cover all the edges of the graph.

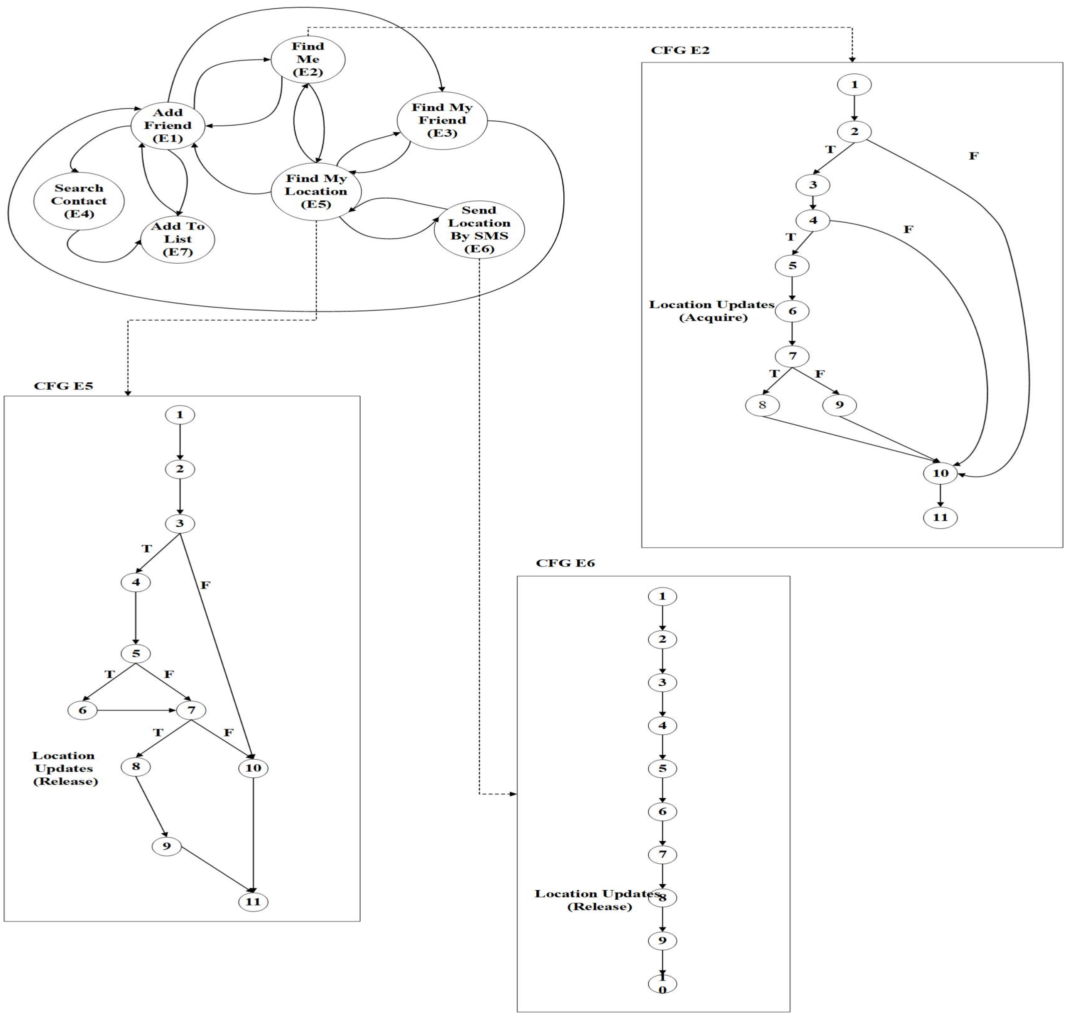
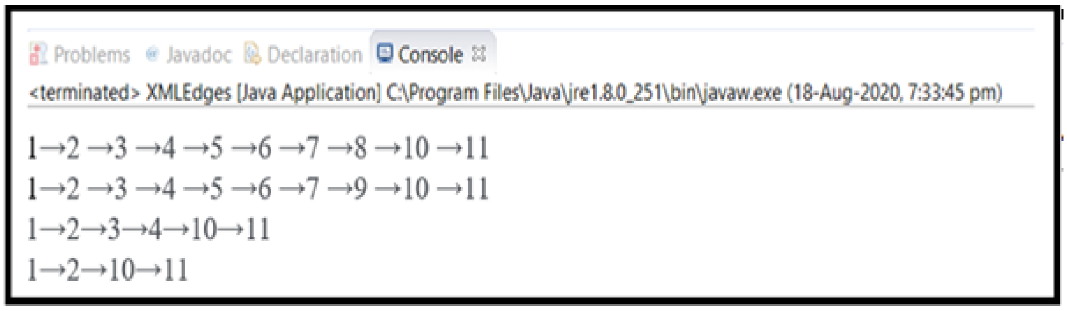


FIGURE 3.6: Control Flow Graphs associated with Events of the EFG in Figure

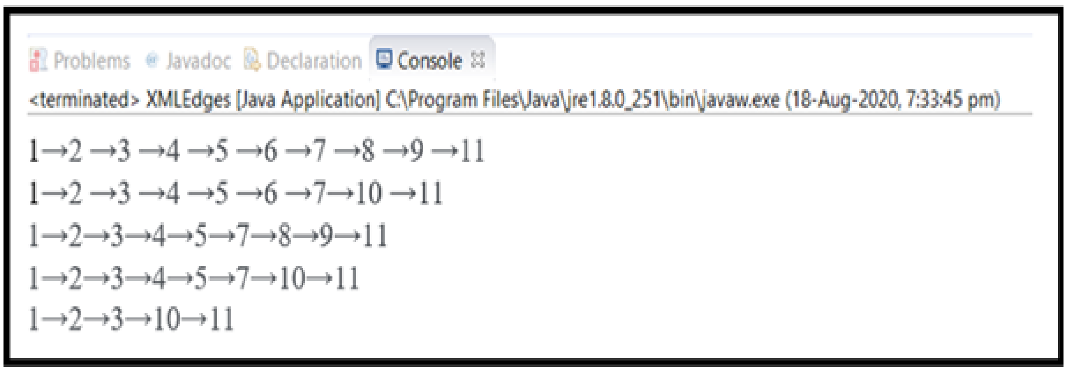
The test-paths generated from the Control Flow Graph of the Event E2 shown in figure 3.6 are shown in figure 3.7.



```
<terminated> XMLEdges [Java Application] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (18-Aug-2020, 7:33:45 pm)
1→2 →3 →4 →5 →6 →7 →8 →10 →11
1→2 →3 →4 →5 →6 →7 →9 →10 →11
1→2→3→4→10→11
1→2→10→11
```

FIGURE 3.7: Test-Paths for the Control Flow Graph of the Event E2

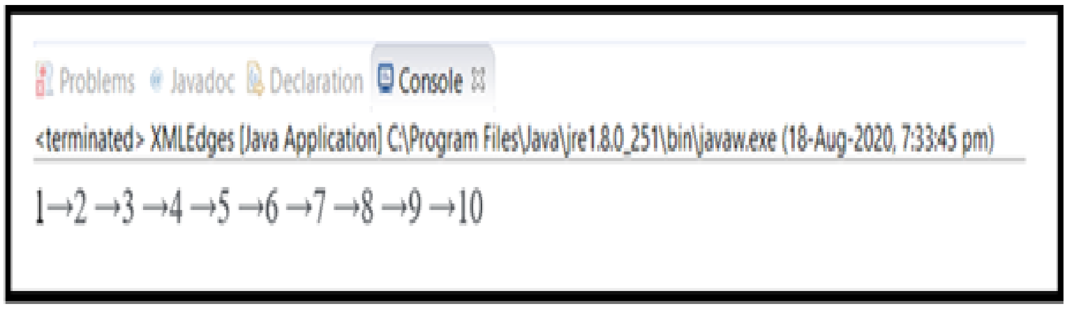
The test-paths generated from the Control Flow Graph of the Event E5 shown in figure 3.6 are shown in figure 3.8.



```
<terminated> XMLEdges [Java Application] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (18-Aug-2020, 7:33:45 pm)
1→2 →3 →4 →5 →6 →7 →8 →9 →11
1→2 →3 →4 →5 →6 →7→10 →11
1→2→3→4→5→7→8→9→11
1→2→3→4→5→7→10→11
1→2→3→10→11
```

FIGURE 3.8: Test-Paths for the Control Flow Graph of the Event E5

The test-paths generated from the Control Flow Graph of the Event E6 shown in Figure 3.6 are shown in Figure 3.9.



```
<terminated> XMLEdges [Java Application] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (18-Aug-2020, 7:33:45 pm)
1→2 →3 →4 →5 →6 →7 →8 →9 →10
```

FIGURE 3.9: Test-Paths for the Control Flow Graph of the Event E6

When the test-paths are generated for each of the Control Flow Graph than we replace these test-paths of each Control Flow Graph with their respective events in the test-paths of the Event Flow Graph to make a complete test-path. In this way, multiple complete test-paths are generated against one path of the Event Flow Graph. We consider only those test-paths of the Event Flow Graph where a corresponding Control Flow Graph is created. Because they are the only events where a resource is being acquired or released in their corresponding event-handlers.

Complete Test-Paths for the Event Flow Graph

Test-Path 1 of EFG:
E1$\rightarrow E2 \rightarrow E3 \rightarrow E7 \rightarrow E3 \rightarrow E2 \rightarrow E5 \rightarrow E1 \rightarrow E8$

We replace the nodes E2 and E5 with the test-paths of their corresponding Control Flow Graphs. After replacement, multiple test-paths are generated against test-path 1 of EFG that are called complete test-paths.

• E1 $\rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow E3 \rightarrow E7 \rightarrow E3 \rightarrow$
$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow$
$7 \rightarrow 8 \rightarrow 9 \rightarrow 11 \rightarrow E1 \rightarrow E8$
• E1 $\rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow E3 \rightarrow E7$
$\rightarrow E3 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 1 \rightarrow$
$2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11 \rightarrow E1 \rightarrow E8$

Test-Path 2 of EFG:
E1$\rightarrow E2 \rightarrow E1 \rightarrow E3 \rightarrow E1 \rightarrow E2 \rightarrow E6 \rightarrow E2 \rightarrow E5 \rightarrow E1 \rightarrow E8$

We replace the nodes E2, E5, and E6 with the test-paths of their corresponding Control Flow Graphs. After replacement, multiple test-paths are generated against test-path 2 of EFG that are called complete test-paths.

• E1 $\rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow E1 \rightarrow E3 \rightarrow E1 \rightarrow$
$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow$
$7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 1 \rightarrow 2 \rightarrow$
$3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow E1 \rightarrow E8$
• E1 $\rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow E1 \rightarrow E3 \rightarrow E1 \rightarrow$
$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow$

$7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11 \rightarrow E1 \rightarrow E8$

Test-Path 3 of EFG:

E1$\rightarrow E4 \rightarrow E8 \rightarrow E1 \rightarrow E2 \rightarrow E5 \rightarrow E2 \rightarrow E5 \rightarrow E1 \rightarrow E8$

We replace the nodes E2 and E5 with the test-paths of their corresponding Control Flow Graphs. After replacement, multiple test-paths are generated against test-path 3 of EFG that are called complete test-paths.

- E1 $\rightarrow E4 \rightarrow E8 \rightarrow E1 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow E1 \rightarrow E8$
- E1 $\rightarrow E4 \rightarrow E8 \rightarrow E1 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5W \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11 \rightarrow E1 \rightarrow E8$

3.1.4 Execution Results/ Detected Bugs

When the complete test-paths are generated, we have to make test-inputs for each test-path. We take each test-path of the Event Flow Graph and generate multiple complete test-paths against it. Once the complete- test-paths are generated, we make test-inputs and evaluate those complete test-paths for the presence of bugs. We make the test-inputs by replacing all the events of the Event Flow Graph and all the test-paths of the Control Flow Graph in the complete test-paths with their corresponding event-name. By considering the Control Flow Graph test-paths only, we saw that there is a bug at CFG E2 because a resource is acquired and not released. However, at the Event Flow Graph level, every resource acquired at each path is also released. These complete test-paths show that the bugs detected in Control Flow Graph are not bugs in the Event Flow Graph but are false positives.

Table 3.1 describes the test-paths of the Event Flow Graph, complete test-paths corresponding to each particular event, and their output.

Table 3.1: Test-Inputs for each Complete Test-Path

EFG Test-Path	**Complete Test-Paths**	**Out-put at CFG Level**	**Out-put at EFG Level**
E1$\to E2 \to E3 \to E7 \to E3 \to E2 \to E5 \to E1 \to E8$	E1$\to 1,2,3,4,5,6,7,8,10,11 \to E3 \to E7 \to E3 \to 1,2,3,4,5,6,7,9,10,11 \to 1,2,3,4,5,6,7,8,9,11 \to E1 \to E8$	Energy-Bug	No bug
	E1$\to 1,2,3,4,5,6,7,8,10,11 \to E3 \to E7 \to E3 \to 1,2,3,4,5,6,7,9,10,11 \to 1,2,3,4,5,7,8,9,11 \to E1 \to E8$	Energy-Bug	No bug
E1$\to E2 \to E1 \to E3 \to E1 \to E2 \to E6 \to E2 \to E5 \to E1 \to E8$	E1$\to 1,2,3,4,5,6,7,8,10,11 \to E1 \to E3 \to E1 \to 1,2,3,4,5,6,7,9,10,11 \to 1,2,3,4,5,6,7,8,9,10 \to 1,2,3,4,5,6,7,8,10,11 \to 1,2,3,4,5,6,7,8,10,11 \to E1 \to E8$	Energy-Bug	No bug
	E1$\to 1,2,3,4,5,6,7,8,10,11 \to E1 \to E3 \to E1 \to 1,2,3,4,5,6,7,9,10,1 \to 1,2,3,4,5,6,7,8,9,10 \to 1,2,3,4,5,6,7,8,10,11 \to 1,2,3,4,5,7,8,9,11 \to E1 \to E8$	Energy-Bug	No bug
E1$\to E4 \to E8 \to E1 \to E2 \to E5 \to E2 \to E5 \to E1 \to E8$	E1$\to E4 \to E8 \to E1 \to 1,2,3,4,5,6,7,8,10,11 \to 1,2,3,4,5,6,7,8,9,11 \to 1,2,3,4,5,6,7,9,10,11 \to E1 \to E8$	Energy Bug	No bug

	E1$\rightarrow$ $E4 \rightarrow E8 \rightarrow E1 \rightarrow 1,2,3,4,5,6,7,8,10,11 \rightarrow 1,2,3,4,5,6,7,8,9,11 \rightarrow 1,2,3,4,5,7,8,9,11 \rightarrow E1 \rightarrow E8$	Energy Bug	No bug

It is concluded that multiple complete test-paths can be generated against every single test-path of the Event Flow Graph. When we generated the test-paths for each Control Flow Graph, we considered each of them in isolation and found that the CFG E2 contains an energy-bug at the method level. The test-paths 1 and 2 of the CFG E2 are buggy paths because a location-update resource is acquired on node 6 but not released anywhere in this method. However, after generating complete test-paths, we found that when the events E5 and E6 occur after the event E1 in the test-path of the Event Flow Graph, the location-update resource is released and there is no energy-bug is detected and therefore, it makes the energy-bug detected at the method level a false-positive at the application level.

Chapter 4

Results and Discussion

This chapter describes the results of the experiments that we have performed on different subject programs and also their results in detail. We have chosen 11 different subject programs from the existing literature and applied our proposed technique on them.

We have generated the Event Flow Graph of each subject program. After generating the Event Flow Graph, we make test-paths of the Event Flow Graph by applying the edge coverage criterion. The edge coverage criterion is a graph coverage criterion that covers all the edges of the graph. These test-paths represent the sequence of events in which they are generated within the application.

We choose those events from the test-paths of the Event Flow Graph whose corresponding event-handler has acquired or released a resource in its code.

We make the Control Flow Graphs of such event-handlers and associate the Control Flow Graphs with their corresponding event in the Event Flow Graph. We make the test-paths of each Control Flow Graph by applying the edge coverage criterion.

When the test-paths are generated for each of the Control Flow Graph, we replace these test-paths with the corresponding event in the test-path of the Event Flow Graph to make a complete test-path.

4.1 Subject Programs

In our experiment, we have used 11 different subject programs. They are discussed below in detail:

1. **Aripuca**

   The aripuca is an Android-based GPS tracking application. It records the track waypoints and the locations in real-time and displays them on Google Maps. It also adds the waypoints, imports and exports data to or from different formats. The problem with this application is that it acquires a location update resource in one of its activities but does not release it within that activity due to which location updates stay on and consumes smartphones' battery that causes an energy-bug [17].

   The source code of the Aripuca application is given in [18] and the apk file can be downloaded from [19].

2. **Tachometer**

   The tachometer is an Android-based application that measures the speed and location and exports the data in a CSV file. It is for use in the car, airplane, train, etc. It is specifically for travelers who want to document the exact location where they stayed. The problem with this application is that when we start to measure the location, it acquires a location update resource in one of its activities but does not release it due to which it consumes smartphones' battery that causes an energy-bug [20].

   The source code of the Aripuca application is given in [21] and the APK file can be downloaded from [22].

3. **Droid-AR**

   Droid-AR is an android based application for Android. The problem with this application is that it causes the GPS to stay on even after the application has been closed by the user.

   The source code of the Droid-AR is given in [23] and the apk file can be downloaded from [24].

4. **Osmdroid**

   Osmdroid is an Android-based application and a replacement for Google's MapView.

   The problem with this application is that it causes the location-updates to stay on even after the application has been closed [25].

   The source code of the Osmdroid is given in [26] and the apk file can be downloaded from [27].

5. **SP-Transport**

   SP-Transport is an Android-based application that helps you to know about the arrival times of buses. It helps you to know about different bus stops. You can also save stops.
   The problem with this application is that it causes the GPS to stay on even after the application is paused [28].

   The source code of the SP-Transport is given in [29] and the apk file can be downloaded from [30].

6. **Ushaidi**

   Ushaidi is an Android-based application that helps you to complete surveys with or without the internet in any location. The problem with this application is that the GPS remains on even after the application has been closed [31].

   The source code of the Ushaidi is given in [32] and the apk file can be downloaded from [33].

7. **Zmanim**

   Zmanim is an Android-based application and tells about Jewish prayer times.

   The problem with this application is that the application acquires a GPS resource and never releases it even after the pause and causes resource leaking [34].

   The source code of the Zmanim is given in [35] and the apk file can be downloaded from [36].

8. **TTS Reader**

   TTS Reader is an Android-based application and helps to convert your text into voice and spoke it out loud.

   The problem with this application is that a wakelock id acquired it but not released [37].

   The apk file can be downloaded from [38].

9. **Better Wifi on/Off**

   Better Wi-Fi on/off is an Android-based application and gives you access to control Wi-Fi state. It uses wakelock and Wi-Fi resources. The problem with this application is that it does not release the wakelock after acquisition [39].

   The source code of the Better Wi-Fi on/off is given in [40] and the apk file can be downloaded from [41].

10. **Fbreader**

    Fbreader is an Android-based eBook application. The problem with this application is that it does not release the wakelock after acquisition [42].

    The source code of the Fbreader is given in [43] and the apk file can be downloaded from [44].

11. **Pedometer**

    Pedometer is an Android-based application and counts the number of steps you walk in a day. It also counts the number of calories that you burn in a day. The problem with this application is that it does not release the wakelock after its acquisition [45].

    The source code of the Pedometer is given in [46] and the apk file can be downloaded from [47].

4.2 Features of the Subject Programs

The features of the subject programs are given in Table 4.1.

Table 4.1: Features of the Subject Programs

App Name	Resource Used	Lines of Code	Event Handler Classes
Aripuca	GPS	8093	15
Tachometer	GPS	793	12
Droid-AR	GPS	18177	6
Osmdroid	GPS	8107	10
SP Transport	GPS	1766	3
Ushaidi	GPS	10621	22
Zmanim	GPS	72977	4
TTS Reader	Wakelock	4560	10
BetterWifi on/Off	Wakelock, Wifi	2926	19
Fbreader	Wakelock	2702	20
Pedometer	Wakelock	783	5

Figure 4.1 represents the graphical representation of lines of code for each subject program.

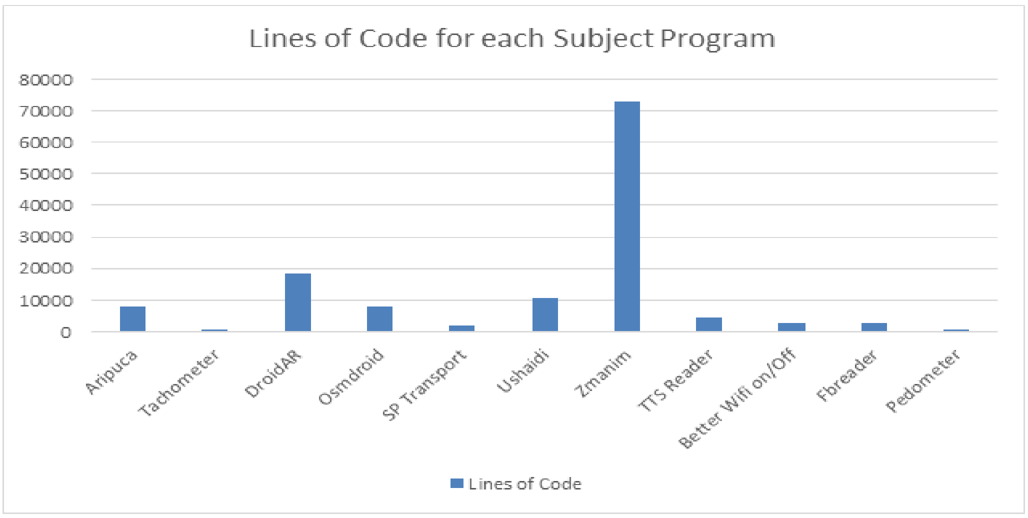


Figure 4.1: Lines of Code for each Subject Program

The above graph shows that the Zmanim application has a greater number of lines of code than the other subject programs.

Figure 4.2 represents the graphical representation of the number of event-handler classes for each subject program.

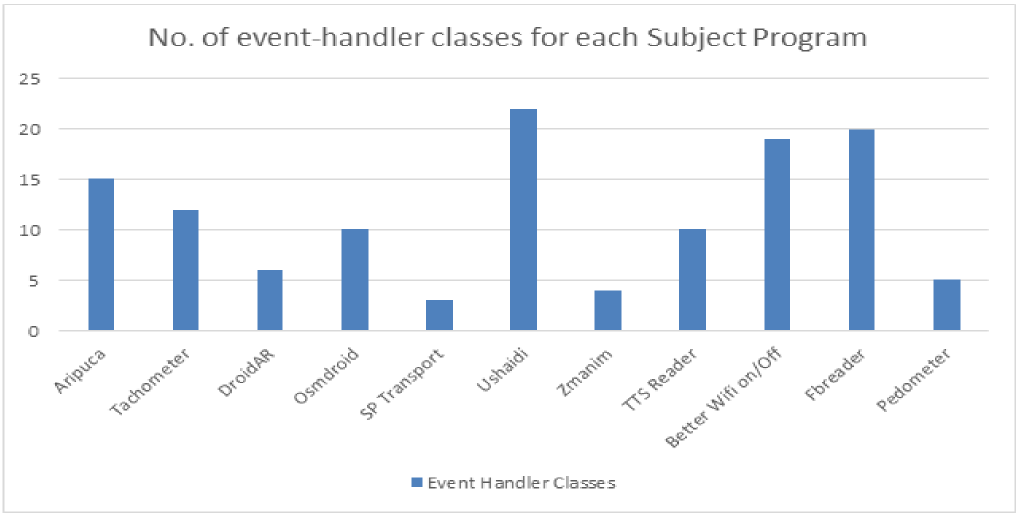


FIGURE 4.2: Number of event-handler Classes for each Subject Program

The above graph shows that the Ushaidi application has a greater number of event-handler classes than the other subject programs.

We have discussed two of the above-mentioned subject programs Aripuca and Tachometer in detail.

The other subject programs are not discussed in detail but their result is mentioned in the result section.

4.3 Event Flow Graph Generation

The Android applications are used to generate the Event Flow Graphs. The Event Flow Graph of each Android application represents its Graphical User Interface.

The Event Flow Graph of the Aripuca and Tachometer is shown in Figure 4.3 and Figure 4.4 respectively.

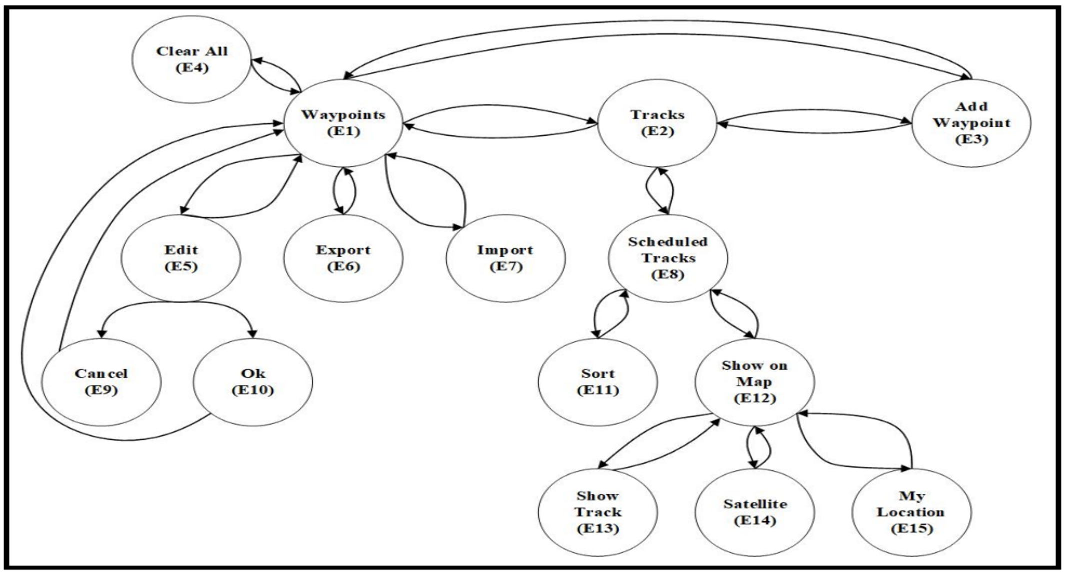


FIGURE 4.3: The Event Flow Graph of Aripuca Application

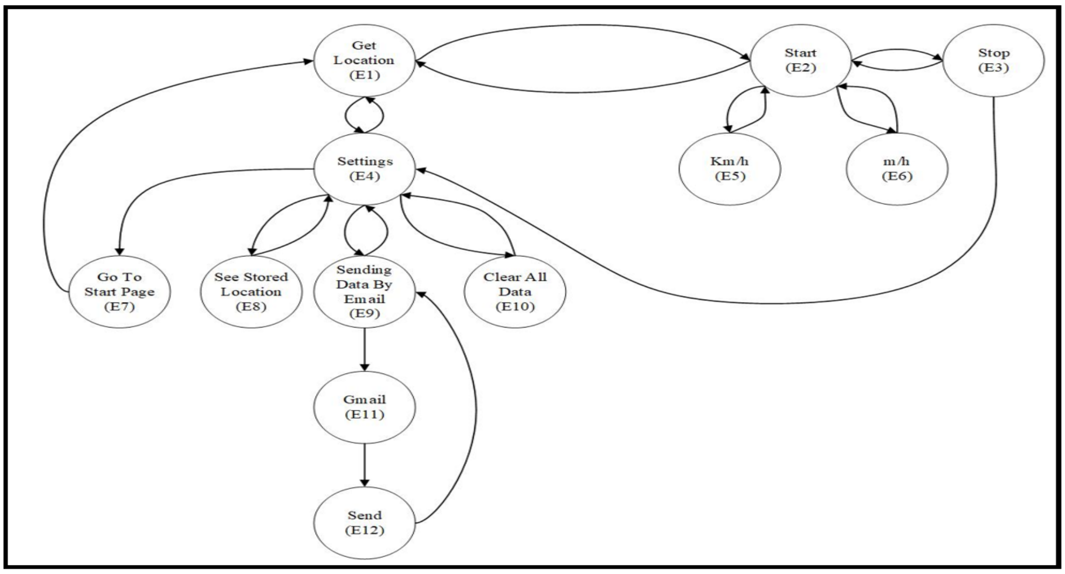


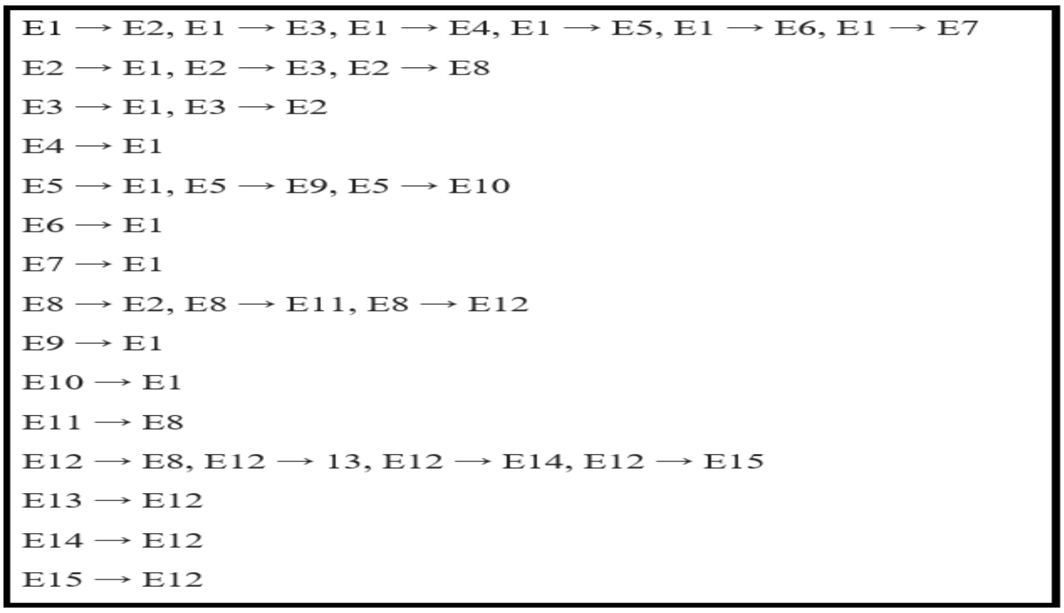
FIGURE 4.4: The Event Flow Graph of Tachometer Application

4.4 Test-Paths of the Event Flow Graph

We have generated the test-paths for the Event Flow Graph of both the Android applications. These test-paths of the Event Flow Graph represent the flow of event-sequences within the application. We have identified all the edges of the Event of Graph of both the Android applications.

When we apply the edge-coverage criterion on both graphs to generate test-paths, we make sure that all these edges are covered by the test-paths that are generated.

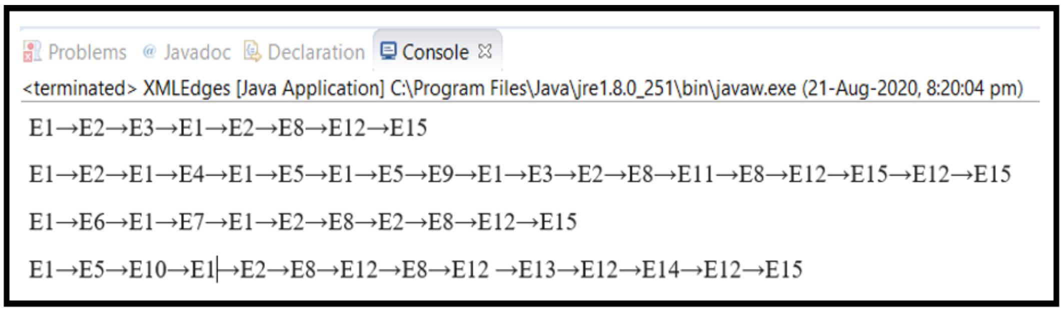
The edges for the Event Flow Graph of Aripuca shown in Figure 4.3 are shown in Figure 4.5:



E1 ⟶ E2, E1 ⟶ E3, E1 ⟶ E4, E1 ⟶ E5, E1 ⟶ E6, E1 ⟶ E7
E2 ⟶ E1, E2 ⟶ E3, E2 ⟶ E8
E3 ⟶ E1, E3 ⟶ E2
E4 ⟶ E1
E5 ⟶ E1, E5 ⟶ E9, E5 ⟶ E10
E6 ⟶ E1
E7 ⟶ E1
E8 ⟶ E2, E8 ⟶ E11, E8 ⟶ E12
E9 ⟶ E1
E10 ⟶ E1
E11 ⟶ E8
E12 ⟶ E8, E12 ⟶ 13, E12 ⟶ E14, E12 ⟶ E15
E13 ⟶ E12
E14 ⟶ E12
E15 ⟶ E12

FIGURE 4.5: Edges of the Event Flow Graph for Aripuca Application

The test-paths for the Event Flow Graph of Aripuca are shown in Figure 4.6. When we applied the edge-coverage criterion on the Event Flow Graph of Aripuca application, a total of four test-paths are generated.



```
<terminated> XMLEdges [Java Application] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (21-Aug-2020, 8:20:04 pm)
E1→E2→E3→E1→E2→E8→E12→E15
E1→E2→E1→E4→E1→E5→E1→E5→E9→E1→E3→E2→E8→E11→E8→E12→E15→E12→E15
E1→E6→E1→E7→E1→E2→E8→E2→E8→E12→E15
E1→E5→E10→E1→E2→E8→E12→E8→E12 →E13→E12→E14→E12→E15
```

FIGURE 4.6: Test-Paths of the Event Flow Graph of Aripuca Application

The edges covered from the test-path 1 are:

E1 $\rightarrow E2, E2 \rightarrow E3, E2 \rightarrow E8, E3 \rightarrow E1, E8 \rightarrow E12, E12 \rightarrow E15$

The edges covered from the test-path 2 are:

E1 $\rightarrow E3, E1 \rightarrow E4, E1 \rightarrow E5, E2 \rightarrow E1, E3 \rightarrow E2, E4 \rightarrow E1, E5 \rightarrow E1, E5 \rightarrow E9, E8 \rightarrow E11, E9 \rightarrow E1, E11 \rightarrow E8, E15 \rightarrow E12$

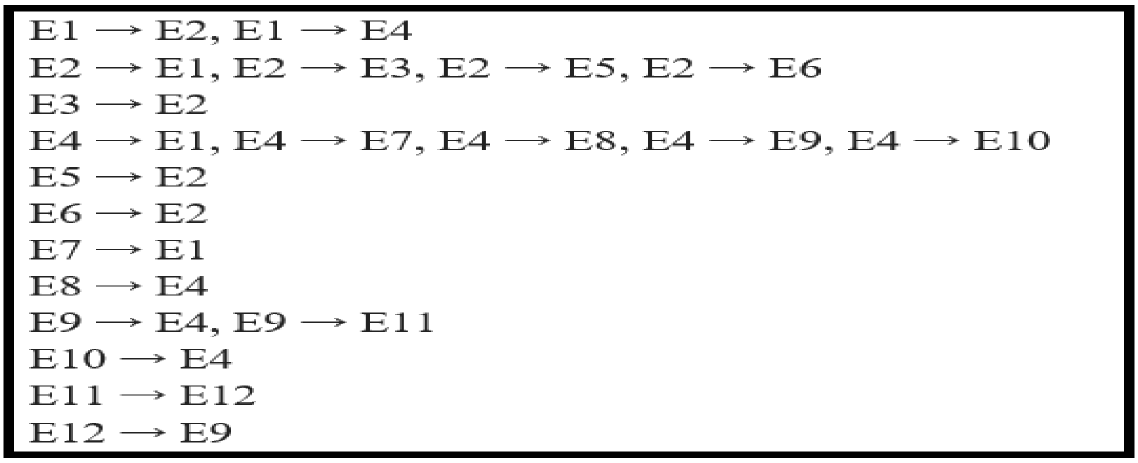
The edges covered from the test-path 3 are:

E1 $\rightarrow E6, E1 \rightarrow E7, E6 \rightarrow E1, E7 \rightarrow E1, E8 \rightarrow E2$

The edges covered from the test-path 4 are:

E5 $\rightarrow E10, E10 \rightarrow E1, E12 \rightarrow E8, E12 \rightarrow E3, E12 \rightarrow E14, E13 \rightarrow E12, E14 \rightarrow E12$

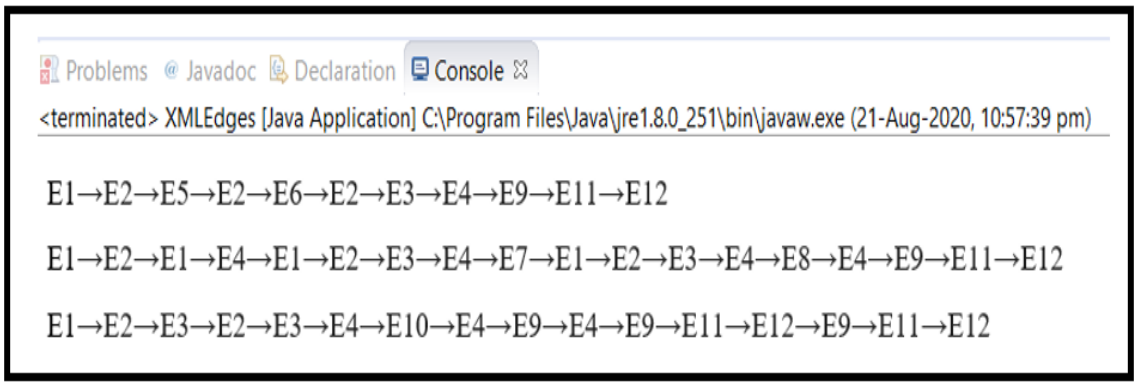
Therefore, all the edges of the Event Flow Graph of the Aripuca application are covered by these test-paths. The edges for the Event Flow Graph of Tachometer shown in Figure 4.4 are shown in Figure 4.7:



```
E1 ⟶ E2, E1 ⟶ E4
E2 ⟶ E1, E2 ⟶ E3, E2 ⟶ E5, E2 ⟶ E6
E3 ⟶ E2
E4 ⟶ E1, E4 ⟶ E7, E4 ⟶ E8, E4 ⟶ E9, E4 ⟶ E10
E5 ⟶ E2
E6 ⟶ E2
E7 ⟶ E1
E8 ⟶ E4
E9 ⟶ E4, E9 ⟶ E11
E10 ⟶ E4
E11 ⟶ E12
E12 ⟶ E9
```

FIGURE 4.7: Edges of the Event Flow Graph for Tachometer Application

The test-paths for the Event Flow Graph of Tachometer are shown in Figure 4.8. When we applied the edge-coverage criterion on the Event Flow Graph of Tachometer application, a total of three test-paths are generated.



```
<terminated> XMLEdges [Java Application] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (21-Aug-2020, 10:57:39 pm)

E1→E2→E5→E2→E6→E2→E3→E4→E9→E11→E12

E1→E2→E1→E4→E1→E2→E3→E4→E7→E1→E2→E3→E4→E8→E4→E9→E11→E12

E1→E2→E3→E2→E3→E4→E10→E4→E9→E4→E9→E11→E12→E9→E11→E12
```

FIGURE 4.8: Test-Paths of the Event Flow Graph of Tachometer Application

The edges covered from the test-path 1 are:

E1 $\rightarrow E2, E2 \rightarrow E3, E2 \rightarrow E5, E2 \rightarrow E6, E3 \rightarrow E4, E4 \rightarrow E9, E5 \rightarrow E2, E6 \rightarrow E2, E11 \rightarrow E12$

The edges covered from the test-path 2 are:

E1 $\rightarrow E4, E2 \rightarrow E1, E4 \rightarrow E1, E4 \rightarrow E7, E4 \rightarrow E8, E7 \rightarrow E1, E8 \rightarrow E4$

The edges covered from the test-path 3 are:

E3 $\rightarrow E2, E4 \rightarrow E10, E9 \rightarrow E4, E10 \rightarrow E4, E12 \rightarrow E9$

Therefore, all the edges of the Event Flow Graph of the Tachometer application are covered by these test-paths.

4.5 Complete Test-Paths

With the help of the test-paths generated from the EFG of both the applications, we have made the complete test-paths. The test-paths of the EFG represent the event sequence in which they execute within the application.

We have chosen those events from these test-paths whose corresponding event-handler method has acquired or released a resource within their code. We have generated the Control Flow Graph of such event-handlers' methods and associated them with their corresponding events in the EFG.

After that, we have generated test-paths of the Control Flow Graphs and replaced those test-paths with their corresponding events that occur in the test-path of the Event Flow Graph to make a complete test-path.

The Control Flow Graph of the event-handlers' methods of the Aripuca application is associated with the Event Flow Graph of the application as shown in Figure 4.9. As described earlier, these are the event-handlers in which a resource is acquired or released.

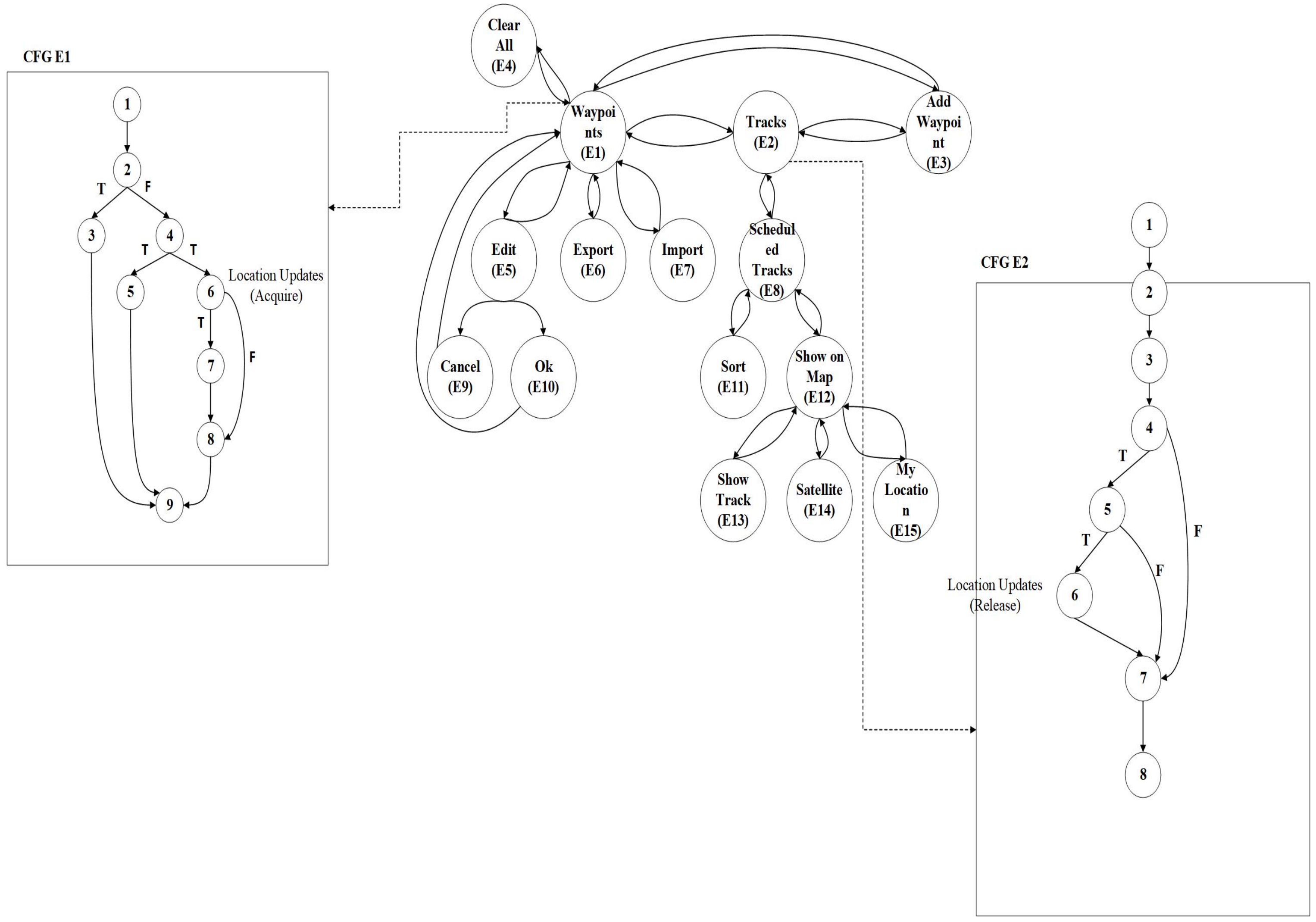
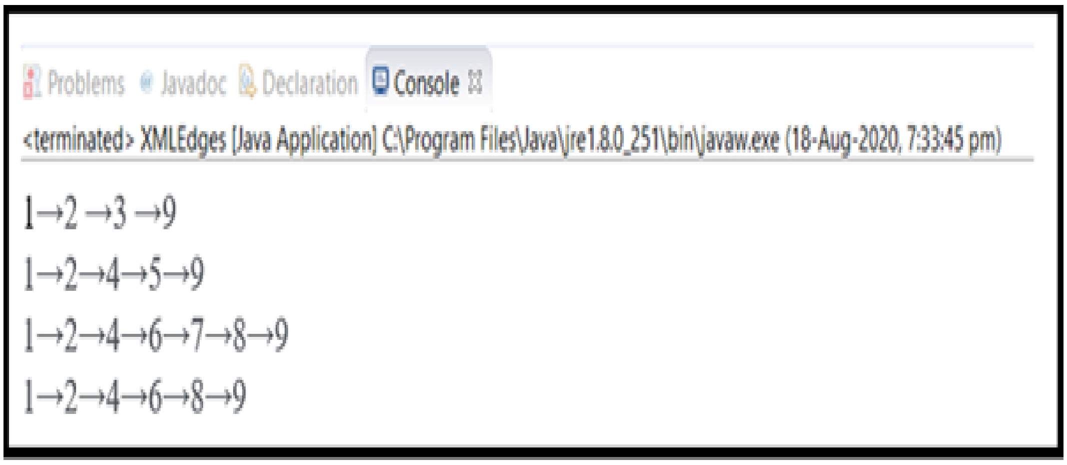


FIGURE 4.9: Control Flow Graphs Associated with Corresponding Events of the Event Flow Graph of Aripuca Application

We have generated the test-paths of the Control Flow Graphs mentioned in Figure 4.9.

The test-paths are generated by using the edge-coverage criterion. The test-paths for the CFG E1 are shown in Figure 4.10.

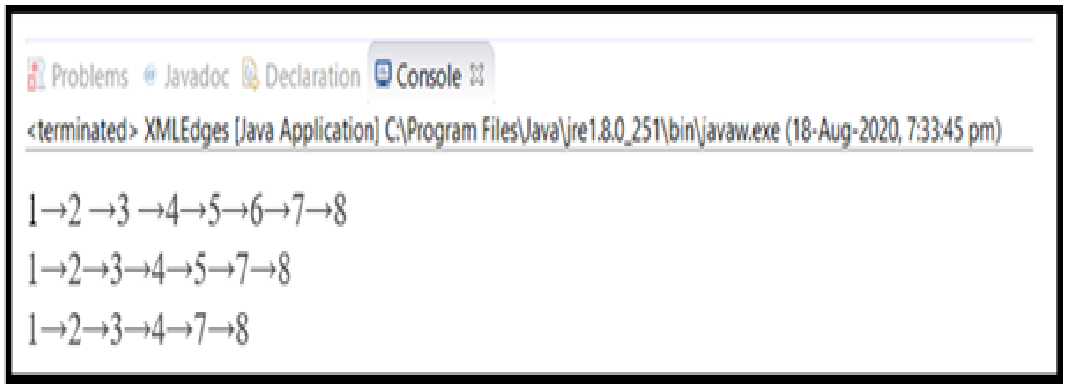


```
<terminated> XMLEdges [Java Application] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (18-Aug-2020, 7:33:45 pm)
1→2 →3 →9
1→2→4→5→9
1→2→4→6→7→8→9
1→2→4→6→8→9
```

FIGURE 4.10: Test-Paths for the CFG E1 of the Aripuca Application

The test-paths containing node 6 are buggy-paths because a resource is acquired at node 6 and is not released. Therefore, test-path 3 and 4 cause an energy-bug.

The test-paths for the CFG E2 are shown in Figure 4.11.



```
<terminated> XMLEdges [Java Application] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (18-Aug-2020, 7:33:45 pm)
1→2 →3 →4→5→6→7→8
1→2→3→4→5→7→8
1→2→3→4→7→8
```

FIGURE 4.11: Test-Paths for the CFG E3 of the Aripuca Application

The Control Flow Graph of the event-handlers of the Tachometer application is associated with the Event Flow Graph of the application as shown in Figure 4.12. As described earlier, these are the event-handlers in which a resource is acquired or released.

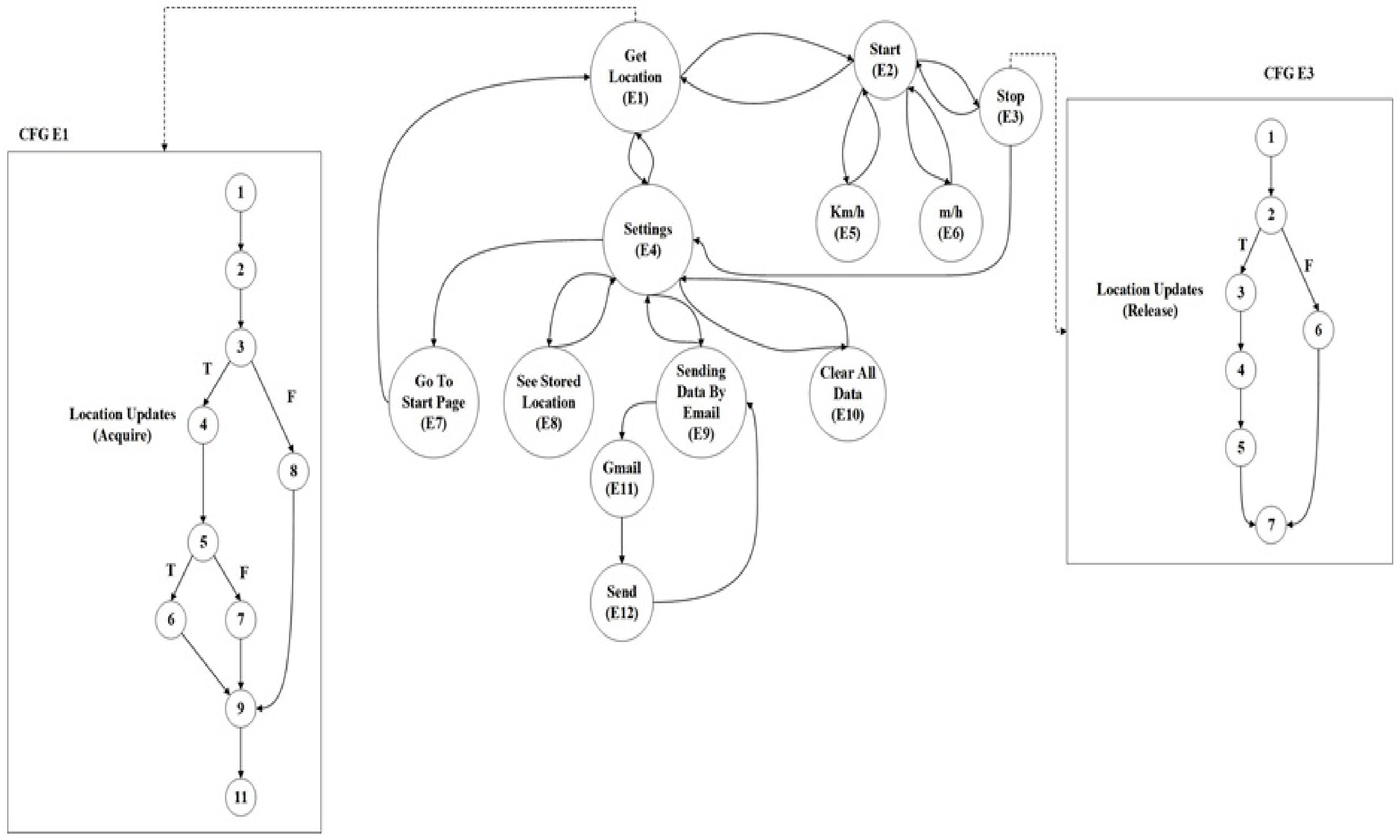
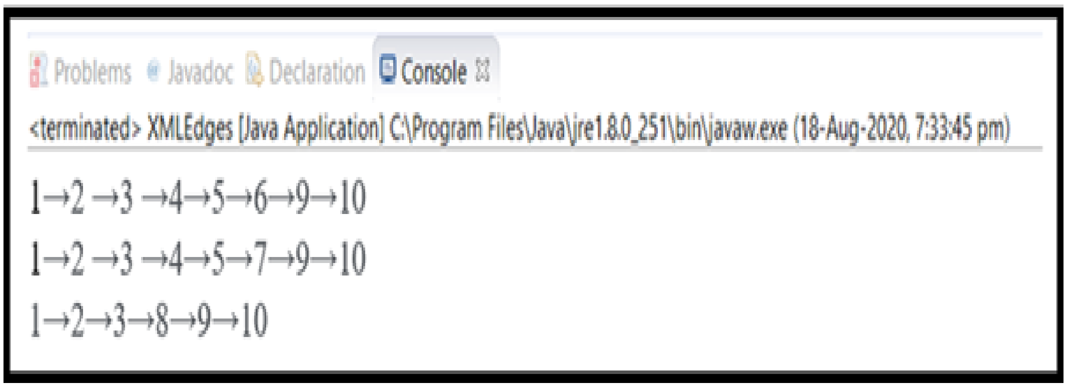


FIGURE 4.12: Control Flow Graphs associated with corresponding events of the Event Flow Graph of Tachometer Application

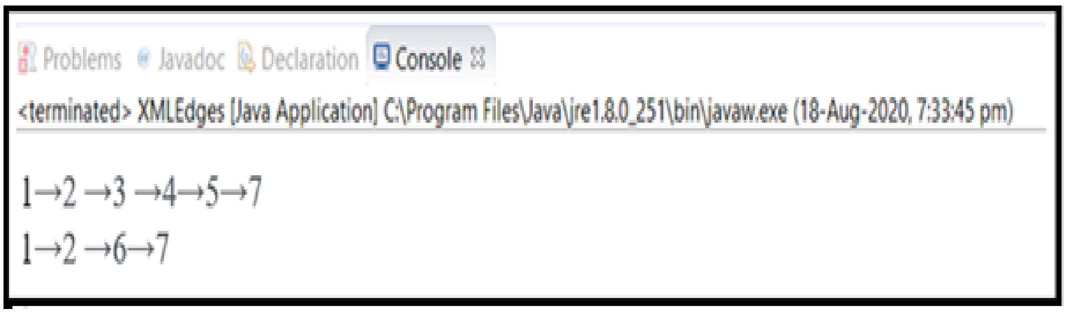
We have generated the test-paths of the Control Flow Graphs mentioned in Figure 4.12. The test-paths are generated by using the edge-coverage criterion. The test-paths for the CFG E1 are shown in Figure 4.13.



```
<terminated> XMLEdges [Java Application] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (18-Aug-2020, 7:33:45 pm)
1→2 →3 →4→5→6→9→10
1→2 →3 →4→5→7→9→10
1→2→3→8→9→10
```

FIGURE 4.13: Test-Paths for the CFG E1 of the Tachometer Application

The test-paths containing node 4 are buggy-paths because a resource is acquired at node 4 and is not released. Therefore test-path 1 and 2 cause an energy-bug. The test-paths for the CFG E2 are shown in Figure 4.14.



```
<terminated> XMLEdges [Java Application] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (18-Aug-2020, 7:33:45 pm)
1→2 →3 →4→5→7
1→2 →6→7
```

FIGURE 4.14: Test-Paths for the CFG E2 of the Tachometer Application

We have replaced these-paths that are generated from the Control Flow Graphs of both the Android applications with their corresponding events that occur in the test-path of their Event Flow Graphs to make a complete test-path.

4.5.1 Complete Test-Paths of the Aripuca Application

For generating the complete test-path of the Aripuca application shown in Figure 4.9, we have replaced the nodes E1 and E2 in the EFG test-paths of the Aripuca application with the test-paths of their corresponding Control Flow Graphs.

4.5.1.1 Test-Paths for Event Flow Graph of Aripuca Application

Test-Path 1: E1$\rightarrow E2 \rightarrow E3 \rightarrow E1 \rightarrow E2 \rightarrow E8 \rightarrow E12 \rightarrow E15$

After replacement, multiple test-paths have been generated against test-path 1 of EFG that are called complete test-paths.

• 1, 2, 4, 6, 7, 8, 9 $\rightarrow 1,2,3,4,5,6,7,8 \rightarrow E3 \rightarrow 1,2,4,6,7,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E15$

• 1, 2, 4, 6, 7, 8, 9 $\rightarrow 1,2,3,4,5,6,7,8 \rightarrow E3 \rightarrow 1,2,4,6,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E15$

• 1, 2, 4, 6, 8, 9 $\rightarrow 1,2,3,4,5,6,7,8 \rightarrow E3 \rightarrow 1,2,4,6,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E15$

• $1,2,4,6,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E3 \rightarrow 1,2,4,6,7,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E15$

Test-Path 2: E1 $\rightarrow E2 \rightarrow E1 \rightarrow E4 \rightarrow E1 \rightarrow E5 \rightarrow E1 \rightarrow E5 \rightarrow E9 \rightarrow E1 \rightarrow E3 \rightarrow E2 \rightarrow E8 \rightarrow E11 \rightarrow E8 \rightarrow E12 \rightarrow E15 \rightarrow E12 \rightarrow E15$

After replacement, multiple test-paths have generated against test-path 2 of EFG that are called complete test-paths.

• 1, 2, 4, 6, 7, 8, 9 $\rightarrow 1,2,3,4,5,6,7,8 \rightarrow 1,2,4,6,8,9 \rightarrow E4 \rightarrow 1,2,4,6,7,8,9 \rightarrow E5 \rightarrow 1,2,4,6,8,9 \rightarrow E5 \rightarrow E9 \rightarrow 1,2,4,6,7,8,9 \rightarrow E3 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E11 \rightarrow E8 \rightarrow E12 \rightarrow E15 \rightarrow E12 \rightarrow E15$

• 1, 2, 4, 6, 7, 8, 9 $\rightarrow 1,2,3,4,5,6,7,8 \rightarrow 1,2,4,6,7,8,9 \rightarrow E4 \rightarrow 1,2,4,6,8,9 \rightarrow E5 \rightarrow 1,2,4,6,8,9 \rightarrow E5 \rightarrow E9 \rightarrow 1,2,4,6,7,8,9 \rightarrow E3 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E11 \rightarrow E8 \rightarrow E12 \rightarrow E15 \rightarrow E12 \rightarrow E15$

Test-Path 3: E1 $\rightarrow E6 \rightarrow E1 \rightarrow E7 \rightarrow E1 \rightarrow E2 \rightarrow E8 \rightarrow E2 \rightarrow E8 \rightarrow E12 \rightarrow E15$

After replacement, multiple test-paths have generated against test-path 3 of EFG that are called complete test-paths.

• 1, 2, 4, 6, 7, 8, 9$\rightarrow E6 \rightarrow 1,2,4,6,8,9 \rightarrow E7 \rightarrow 1,2,4,6,7,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E15$

• 1, 2, 4, 6, 8, 9$\rightarrow E6 \rightarrow 1,2,4,6,7,8,9 \rightarrow E7 \rightarrow 1,2,4,6,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E15$

Test-Path 4: E1$\rightarrow E5 \rightarrow E10 \rightarrow E1 \rightarrow E2 \rightarrow E8 \rightarrow E12 \rightarrow E8 \rightarrow E12 \rightarrow E13 \rightarrow E12 \rightarrow E14 \rightarrow E12 \rightarrow E15$

After replacement, multiple test-paths have generated against test-path 4 of EFG that are called complete test-paths.

• 1, 2, 4, 6, 7, 8, 9$\rightarrow E5 \rightarrow E10 \rightarrow 1,2,4,6,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E8 \rightarrow E12 \rightarrow E13 \rightarrow E12 \rightarrow E14 \rightarrow E12 \rightarrow E15$
• 1, 2, 4, 6, 8, 9$\rightarrow E5 \rightarrow E10 \rightarrow 1,2,4,6,7,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E8 \rightarrow E12 \rightarrow E13 \rightarrow E12 \rightarrow E14 \rightarrow E12 \rightarrow E15$

4.5.2 Complete Test-Paths of the Tachometer Application

For generating the complete test-path of the Tachometer application shown in Figure 4.12.

We have replaced the nodes E1 and E3 in the EFG test-paths of the Aripuca application with the test-paths of their corresponding Control Flow Graphs.

4.5.2.1 Test-Paths for Event Flow Graph of Tachometer Application

Test-Path 1: E1 $\rightarrow E2 \rightarrow E5 \rightarrow E2 \rightarrow E6 \rightarrow E2 \rightarrow E3 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12$

After replacement, multiple test-paths have been generated against test-path 1 of EFG that are called complete test-paths.

• 1, 2, 3, 4, 5, 6, 9, 10 $\rightarrow E2 \rightarrow E5 \rightarrow E2 \rightarrow E6 \rightarrow E2 \rightarrow 1,2,3,4,5,7 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12$
• 1, 2, 3, 4, 5, 7, 9, 10 $\rightarrow E2 \rightarrow E5 \rightarrow E2 \rightarrow E6 \rightarrow E2 \rightarrow 1,2,3,4,5,7 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12$

Test-Path 2: E1 $\rightarrow E2 \rightarrow E1 \rightarrow E4 \rightarrow E1 \rightarrow E2 \rightarrow E3 \rightarrow E4 \rightarrow E7 \rightarrow E1 \rightarrow E2 \rightarrow E3 \rightarrow E4 \rightarrow E8 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12$

After replacement, multiple test-paths have generated against test-path 2 of EFG that are called complete test-paths.

- 1, 2, 3, 4, 5, 6, 9, 10 $\rightarrow E2 \rightarrow 1, 2, 3, 4, 5, 7, 9, 10 \rightarrow E4 \rightarrow 1, 2, 3, 4, 5, 6, 9, 10 \rightarrow E2 \rightarrow 1, 2, 3, 4, 5, 7 \rightarrow E4 \rightarrow E7 \rightarrow 1, 2, 3, 4, 5, 7, 9, 10 \rightarrow E2 \rightarrow 1, 2, 3, 4, 5, 7 \rightarrow E4 \rightarrow E8 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12$
- 1, 2, 3, 4, 5, 7, 9, 10 $\rightarrow E2 \rightarrow 1, 2, 3, 4, 5, 6, 9, 10 \rightarrow E4 \rightarrow 1, 2, 3, 4, 5, 7, 9, 10 \rightarrow E2 \rightarrow 1, 2, 3, 4, 5, 7 \rightarrow E4 \rightarrow E7 \rightarrow 1, 2, 3, 4, 5, 7, 9, 10 \rightarrow E2 \rightarrow 1, 2, 3, 4, 5, 7 \rightarrow E4 \rightarrow E8 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12$

Test-Path 3: E1 $\rightarrow E2 \rightarrow E3 \rightarrow E2 \rightarrow E3 \rightarrow E4 \rightarrow E10 \rightarrow E4 \rightarrow E9 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12 \rightarrow E9 \rightarrow E11 \rightarrow E12$

After replacement, multiple test-paths have generated against test-path 3 of EFG that are called complete test-paths.

- 1, 2, 3, 4, 5, 6, 9, 10 $\rightarrow E2 \rightarrow 1, 2, 3, 4, 5, 7 \rightarrow E2 \rightarrow 1, 2, 3, 4, 5, 7 \rightarrow E4 \rightarrow E10 \rightarrow E4 \rightarrow E9 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12 \rightarrow E9 \rightarrow E11 \rightarrow E12$
- 1, 2, 3, 4, 5, 7, 9, 10 $\rightarrow E2 \rightarrow 1, 2, 3, 4, 5, 7 \rightarrow E2 \rightarrow 1, 2, 3, 4, 5, 7 \rightarrow E4 \rightarrow E10 \rightarrow E4 \rightarrow E9 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12 \rightarrow E9 \rightarrow E11 \rightarrow E12$

4.6 Test-Paths Evaluation

In this section, we described those test-inputs that we have generated against the each complete test-path.

After generating the test-inputs, we have evaluated the complete test-paths both on CFG and EFG level.

The complete test-paths against one single test-path of the Event Flow Graph of the Aripuca application. The evaluation of this particular application is shown in the Table 4.2.

TABLE 4.2: Test-Inputs for a Complete Test-Path of the Aripuca EFG

EFG Test-Path	Complete Test-Paths	Output at CFG Level	Output at EFG Level
E1 $\rightarrow E2 \rightarrow E3 \rightarrow E1 \rightarrow E2 \rightarrow E8 \rightarrow E12 \rightarrow E15$	1, 2, 4, 6, 7, 8, 9 $\rightarrow 1,2,3,4,5,6,7,8 \rightarrow E3 \rightarrow 1,2,4,6,7,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E15$	Energy-Bug	No Bug
	1, 2, 4, 6, 7, 8, 9 $\rightarrow 1,2,3,4,5,6,7,8 \rightarrow E3 \rightarrow 1,2,4,6,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E15$	Energy-Bug	No Bug
	1, 2, 4, 6, 8, 9 $\rightarrow 1,2,3,4,5,6,7,8 \rightarrow E3 \rightarrow 1,2,4,6,8,\rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E15$	Energy-Bug	No Bug
	1, 2, 4, 6, 8, 9 $\rightarrow 1,2,3,4,5,6,7,\rightarrow E3 \rightarrow 1,2,4,6,7,8,9 \rightarrow 1,2,3,4,5,6,7,8 \rightarrow E8 \rightarrow E12 \rightarrow E15$	Energy-Bug	No Bug

It is concluded from Table 4.2 that 4 test-paths were generated for the Event Flow Graph of the Aripuca application. Four (4) complete test-paths were generated against test-path 1 of the Event Flow Graph. Two (2) complete test-paths were generated against each test-path 2, 3, and four (4) of the Event Flow Graph as shown in Figure 4.15.

The test-paths 1 and 2 of the CFG E1 are buggy paths because a location-update resource is acquired on node 4 but not released anywhere in this method. However, after generating complete test-paths, we found that when the event E3 occurs after the event E1 in the test-path of the Event Flow Graph, the location-update resource is released and there is no energy-bug is detected and therefore, it makes the energy-bug detected at the method level a false-positive at the application level. The complete test-paths against one single test-path of the Event Flow Graph of the Tachometer application, and its evaluation are shown in Table 4.3.

Table 4.3: Test-Inputs for a Complete Test-Path of the Tachometer EFG

EFG Test-Path	**Complete Test-Paths**	**Output at CFG Level**	**Output at EFG Level**
E1$\rightarrow E2 \rightarrow E5 \rightarrow E2 \rightarrow E6 \rightarrow E2 \rightarrow E3 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12$	1,2,3,4,5,6,9,10$\rightarrow E2 \rightarrow E5 \rightarrow E2 \rightarrow E6 \rightarrow E2 \rightarrow 1,2,3,4,5,7 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12$	Energy-Bug	False-Positive
	1,2,3,4,5,7,9,10$\rightarrow E2 \rightarrow E5 \rightarrow E2 \rightarrow E6 \rightarrow E2 \rightarrow 1,2,3,4,5,7 \rightarrow E4 \rightarrow E9 \rightarrow E11 \rightarrow E12$	Energy-Bug	False-Positive

It is concluded from Table 4.3 that 3 test-paths were generated for the Event Flow Graph of the Tachometer application.

2 complete test-paths were generated against each test-path of the Event Flow Graph as shown in Figure 4.15.

The test-paths 3 and 4 (two test-paths) of the Control Flow Graph E1 are buggy paths because a location-update resource is acquired on node 4 but not released anywhere.

However, after generating complete test-paths, we found that when the event E2 occurs after the event E1 in the test-path of the Event Flow Graph, the location-update resource is released and there is no energy-bug is detected.

Therefore, it makes the energy-bug detected at the method level a false-positive at the application level.

4.7 Results

We have chosen 11 real-life Android applications from online sources such as GitHub and Google Play to perform experiments. These applications that we have used in our experiments have at least 700 lines-of-code and use energy-intensive resources.

Our proposed technique reported false-positives for 10 out of 11 applications.

Table 4.4 describes the details include the total EFG test-paths, complete test-paths, and bug-free paths in each subject program.

Table 4.4: No. of EFG Test-Paths, no. of complete test-paths, and bug-free paths in each Subject Program

App Name	**EFG Test-Paths**	**Complete Test-Paths**	**Bug-Free Paths**
Aripuca	4	10	10
Tachometer	3	6	6
Droid-AR	2	4	4
Osmdroid	6	11	11
SP Transport	2	4	4
Ushaidi	8	14	14
Zmanim	2	4	4
TTS Reader	5	8	8
BetterWifi on/Off	6	10	10
Fbreader	7	12	12
Pedometer	4	8	6

The graphical representation of Table 4.4 is shown in Figure 4.15.

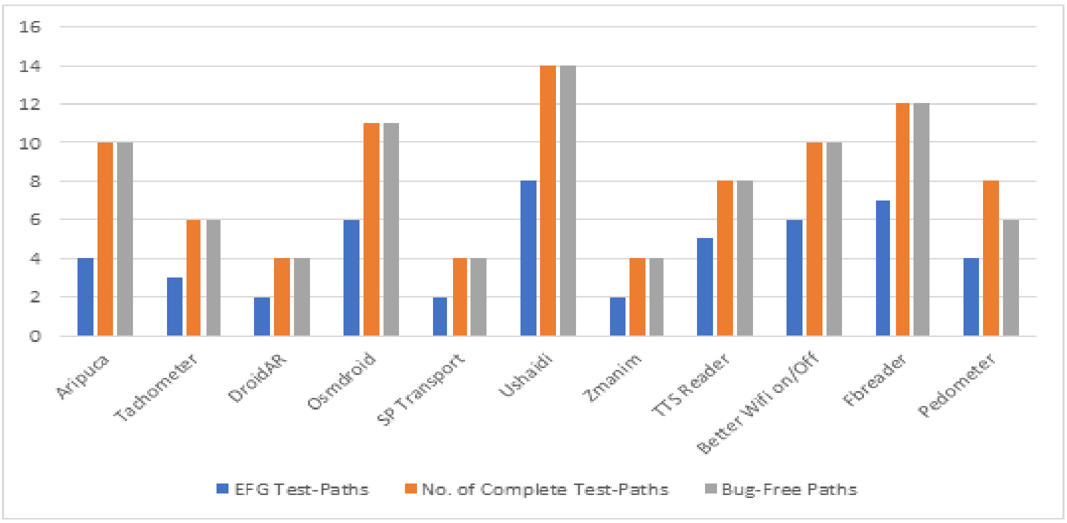


FIGURE 4.15: No. of EFG Test-Paths and no. of complete test-paths in each Subject Program

The above graph represents number of EFG test-paths and number of complete test-paths generated from each subject program. A greater number of EFG test-paths and number of false-positives were detected in Ushaidi application. Droid-AR, SP-Transport and Zmanim generated equal number of EFG test-paths and number of complete test-paths. Pedometer is the only subject program that contains 2 real energy-bugs. All other subject programs contain no energy-bug. From the above graph, it is concluded that among these 11 Android applications, 7 have used GPS (location update) resource, 3 have used wakelock resource and 1 have used both wakelock and Wifi resource. Zmanim application has more lines-of-code than other applications and Ushaidi has a greater number of event-handler classes than other applications. All applications have used at least one energy-intensive resource.

4.8 Comparison

We have chosen 11 Android applications and compared them with proposed technique and technique [3]. On inspecting applications for detection of energy-bugs, we observed that a resource is acquired in an event-handler but not released; that case makes it an energy-bug at the method level. However, on inspecting further, we found that particular resource is released in some other event-handler and that makes it FPs at application level. The comparison is given in Table 4.5.

TABLE 4.5: Comparison of Proposed Approach with Existing Approach

App Name	**Energy-Bugs detected by Banerjee et al. [3]**	**False-Positives detected by Proposed Approach**	**Real Energy Bugs in each Application**
Aripuca	2 energy-bugs were detected	2 false-positives were detected	No energy bugs
Tachometer	2 energy-bugs were detected	2 false-positives were detected	No energy bugs
Droid-AR	1 energy-bugs were detected	1 false-positives were detected	No energy bugs
Osmdroid	3 energy-bugs were detected	3 false-positives were detected	No energy bugs
SP Transport	1 energy-bugs were detected	1 false-positives were detected	No energy bugs
Ushaidi	4 energy-bugs were detected	4 false-positives were detected	No energy bugs
Zmanim	1 energy-bugs were detected	1 false-positives were detected	No energy bugs
TTS Reader	7 energy-bugs were detected	7 false-positives were detected	No energy bugs
Better Wifi on/Off	12 energy-bugs were detected	12 false-positives were detected	No energy bugs
Fbreader	14 energy-bugs were detected	14 false-positives were detected	No energy bugs
Pedometer	2 energy-bugs were detected	No false-positives were detected	2 energy bugs

The above table concludes that Pedometer is the only application that contains 2 real energy bugs and our proposed approach also detected those energy bugs. On inspecting the applications for the detection of energy-bugs, we observed that a resource is acquired in an event-handler but not released; that case makes it an energy-bug at the method level. However, on inspecting further, we found that the particular resource is released in some other event-handler and that makes it false-positive at the application level. It is also concluded that the proposed approach only detects the real energy bugs that exist in an application and eliminates false positives whereas the existing approach [3] did not detect the real energy bugs.

Figure 4.16 shows the graphical representation of Table 4.5.

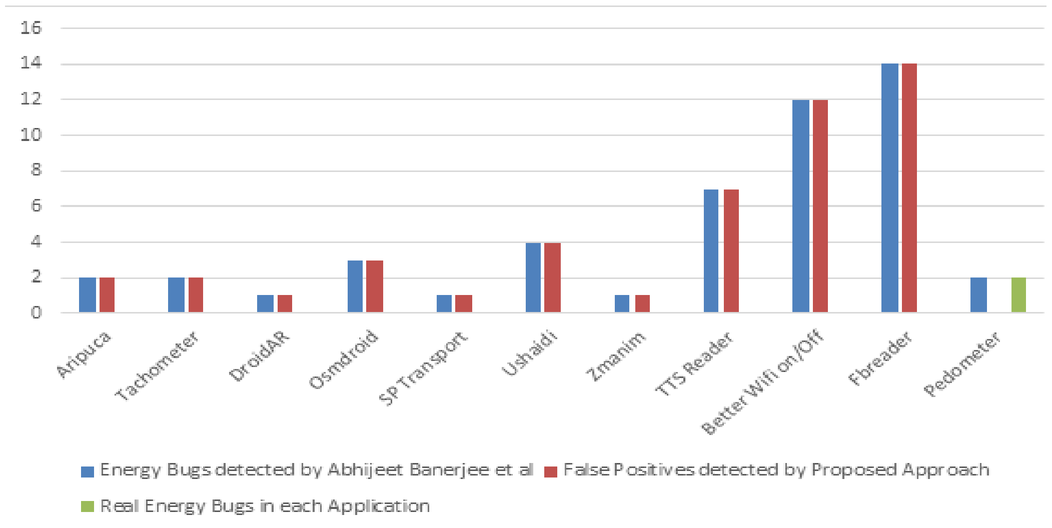


FIGURE 4.16: Comparison of Proposed Approach with Existing Approach

It is concluded that the energy-bugs detected at the method level by Banerjee et al [3] are FPs at application level and proposed approach detected those FPs. Proposed approach detected real energy-bugs only in Pedometer application because it contains real energy-bugs. The existing approach has used symbolic execution for energy bug detection whereas proposed approach performs actual execution of code to detect energy bugs. Symbolic execution cannot identify infeasible paths whereas when code is executed then infeasible paths are excluded as we have generated paths by using a graph coverage criterion. Therefore, cost for actual execution may increase but this will also help in detecting energy bugs from large-scale applications. However, symbolic execution is not useful for large scale applications.

Chapter 5

Conclusion and Future work

In this research, we have proposed a dynamic technique to detect the presence of energy-bugs in Android applications. We have performed the energy-bug detection at the application level. The use of Event Flow Graph for each application helps us to identify the energy-bugs at the application level. The existing techniques worked on the method level bug detection. They focus on the methods in isolation and if a resource is acquired in a method and not released, they call it an energy-bug. However, at the application level if a resource is acquired in one method and released in another method, such a path may execute in the test-paths of the Event Flow Graph that execute both the methods and releases that resource generating no energy-bug at the application level and generates false-positives in the existing technique. Therefore, energy-bugs detected at the method level may be false-positives at the application level.

RQ1: What are the gaps in the existing techniques?

From the literature survey, we have identified several techniques that can detect energy-bugs in an Android application. Some of them use dynamic analysis techniques to detect energy-bugs and some use a hybrid approach. In dynamic analysis techniques, they use dynamic taint-tracking to detect energy-bugs. Some of the literature studies monitor the applications during their execution and detect which resources are left unreleased at the end of the execution. In static analysis techniques, they statically analyze the program code and identify which resources are left unreleased and causing energy-bug. They measure the power consumption of the smartphone due to buggy applications by attaching a power-meter with the device. Hybrid approaches are also statically analyzing the presence of energy-bugs by analyzing which resources are left unreleased. The existing techniques work on method level energy-bug detection and consider the methods in isolation. They do not focus on application level bug detection and generate false-positives.

RQ2: How to enhance the existing techniques for application level energy-bug detection?
If a resource is acquired in one method and released in another method. When both of these methods execute in the test-path of an Event Flow Graph than the resource is released and it does not cause an energy-bug. However, if we consider both the methods in isolation than this is an energy-bug because the resource is not released after acquisition. Therefore, the proposed technique enhances the existing technique by detecting the false-positives in the existing technique.

RQ3: Is the proposed technique better in terms of eliminating false-positives?
The energy-bugs detected with the existing technique at the method level are not actual bugs because our proposed approach improves the energy-bug detection rate by eliminating the false-positives in the existing technique and is better than the existing technique.

5.1 Future Work

By using our proposed approach, several research directions can be studied in the future. In particular, there are several other coverage criteria for the graph coverage such as loop coverage. We plan to apply other coverage criterion on the Control Flow Graph and the Event Flow Graph. They can generate more test-paths and can be experimented in the future. Strong coverage criterion means a greater number of test-paths and this increase the cost. We can make our approach hybrid in future as this will minimize the cost.

Bibliography

[1] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in android applications," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 389–398.

[2] 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, pp. 588–598.

[3] 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*, vol. 44, no. 5, pp. 470–490, 2017.

[4] Y. Liu, C. Xu, and S.-C. Cheung, "Where has my battery gone? finding sensor related energy black holes in smartphone applications," in *2013 IEEE international conference on pervasive Computing and Communications (PerCom)*. IEEE, 2013, pp. 2–10.

[5] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, "Light-weight, inter-procedural and callback-aware resource leak detection for android apps," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1054–1076, 2016.

[6] H. Wu, S. Yang, and A. Rountev, "Static detection of energy defect patterns in android applications," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 185–195.

[7] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. Dinda, and L. Yang, "Adel: An automatic detector of energy leaks for smartphone applications," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2012, pp. 363–372.

[8] Y. Liu, C. Xu, S.-C. Cheung, and J. Lü, "Greendroid: Automated diagnosis of energy inefficiency for smartphone applications," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 911–940, 2014.

[9] A. M. Abbasi, M. Al-tekreeti, Y. Ali, K. Naik, A. Nayak, N. Goel, and B. Plourde, "A framework for detecting energy bugs in smartphones," in *2015 6th International Conference on the Network of the Future (NOF)*. IEEE, 2015, pp. 1–3.

[10] J. Wang, Y. Liu, C. Xu, X. Ma, and J. Lu, "E-greendroid: effective energy inefficiency analysis for android applications," in *Proceedings of the 8th Asia-Pacific Symposium on Internetware*, 2016, pp. 71–80.

[11] Q. Li, C. Xu, Y. Liu, C. Cao, X. Ma, and J. Lü, "Cyandroid: stable and effective energy inefficiency diagnosis for android apps," *Science China Information Sciences*, vol. 60, no. 1, p. 012104, 2017.

[12] Y. Liu, J. Wang, C. Xu, and X. Ma, "Navydroid: detecting energy inefficiency problems for smartphone applications," in *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, 2017, pp. 1–10.

[13] A. M. Abbasi, M. Al-Tekreeti, K. Naik, A. Nayak, P. Srivastava, and M. Zaman, "Characterization and detection of tail energy bugs in smartphones," *IEEE Access*, vol. 6, pp. 65 098–65 108, 2018.

[14] Arif. M-Memon, "An event-flow model of GUI-based applications for testing," *Software testing, verification and reliability*, vol. 17, pp. 3 137–157 108, 2007.

[15] A. Developers, *UI/Application Exerciser Monkey*, 2013 (accessed August 25, 2020). [Online]. Available: https://developer.android.com/studio/test/monkey

[16] M. Danilo-Bruschi, “Detecting Self-Mutating Malware Using Control Flow Graph Matching,” *URl: Citeseerx.ist.psu.edu*, *Springer*, vol. 17, pp. 3 129–143 108, 2006.

[17] VitalNik, *Aripuca GPS Tracker*, (accessed September 04, 2020). [Online]. Available: https://play.google.com/store/apps/details?id=com.aripuca.tracker&hl=en

[18] Vitanik, *Github:desword/aripuca-tracker*, (accessed September 04, 2020). [Online]. Available: https://github.com/desword/aripuca-tracker

[19] vitaNik, *Apkfun:Aripuca GPS Trackerr*, (accessed September 04, 2020). [Online]. Available: https://apkpure.com/aripuca-gps-tracker/com.aripuca.tracker

[20] Androcalc, *Tachometer*, (accessed September 04, 2020). [Online]. Available: https://play.google.com/store/apps/details?id=com.diogines.tachometer&hl=en

[21] androcalc, *Github: Polymer/tachometer*, (accessed September 04, 2020). [Online]. Available: https://github.com/Polymer/tachometer

[22] aandrocalc, *Apkpure: tachometer*, (accessed September 04, 2020). [Online]. Available: https://apkpure.com/tachometer/com.diogines.tachometer

[23] Droider, *Google Play Store:droidar - default*, (accessed September 04, 2020). [Online]. Available: https://code.google.com/archive/p/droidar/source/default/source

[24] L. O. T. A. Variety, *Download DroidAR (Car Finder) APK*, (accessed September, 2020). [Online]. Available: https://apk.fun/de.rwth.html

[25] N. Boyd, *osmdroid*, (accessed September, 2020). [Online]. Available: https://play.google.com/store/apps/details?id=org.osmdroid&hl=en

[26] G. N. Boyd, *osmdroid/osmdroid*, (accessed September, 2020). [Online]. Available: https://github.com/osmdroid/osmdroid

[27] N. boyd, *Osmdroid*, (accessed September 04, 2020). [Online]. Available: https://apkpure.com/osmdroid/org.osmdroid

[28] T. Halvadzhiev, *Google Play Store: Sofia Public Transport*, (accessed September, 2020). [Online]. Available: https://play.google.com/store/apps/details?id=com.bearenterprises.sofiatraffic&hl=en

[29] T. halvadzhiev, *Github:SourceCodeBackup/sofia-public-transport-navigator*, (accessed September 04, 2020). [Online]. Available: https://github.com/SourceCodeBackup/sofia-public-transport-navigator

[30] teodor halvadzhiev, *Apkpure: Sofia Public Transport*, (accessed September, 2020). [Online]. Available: https://apkpure.com/sofia-public-transport/com.bearenterprises.sofiatraffic

[31] U. Inc., *Google Paly Store: Ushahidi*, (accessed September, 2020). [Online]. Available: https://play.google.com/store/apps/details?id=com.ushahidi.mobile&hl=en_AU

[32] ushahidi Inc., *Github: Ushahidi*, (accessed September, 2020). [Online]. Available: https://github.com/ushahidi/Ushahidi

[33] U. Inc., *apkpure: Ushahidi Classic*, (accessed September 04, 2020). [Online]. Available: https://apkpure.com/ushahidi-classic/com.ushahidi.android.app

[34] J. Gindin, *Google Play Store: Zmanim*, (accessed September, 2020). [Online]. Available: https://play.google.com/store/apps/details?id=com.gindin.zmanim.android&hl=en

[35] jay Gindin, *Github: KosherJava/zmanim*, (accessed September 04, 2020). [Online]. Available: https://github.com/KosherJava/zmanim

[36] jay gindin, *Apkpure: zmanim*, (accessed September 04, 2020). [Online]. Available: https://apkpure.com/zmanim/com.gindin.zmanim.android

[37] D. Nachmani, *Google play store:TTS Reader*, (accessed September, 2020). [Online]. Available: https://play.google.com/store/apps/details?id=com.davidnac.ttsreaderfree&hl=en

[38] Libera, *apkpure:TTS Reader reads aloud books, all books*, (accessed September 04, 2020). [Online]. Available: https://play.google.com/store/apps/details?id=com.davidnac.ttsreaderfree&hl=en

[39] S. Knispel, *Google Play Store: Better Wifi On/Off*, (accessed September 04, 2020). [Online]. Available: https://play.google.com/store/apps/details?id=com.asksven.betterwifionoff&hl=en

[40] Asksven, *Github: asksven/BetterWifiOnOffl*, (accessed September 04, 2020). [Online]. Available: https://github.com/asksven/BetterWifiOnOff?files=1

[41] S. KnispeL, *apkpure:Better Wifi On/Off*, (accessed September 04, 2020). [Online]. Available: https://apkpure.com/better-wifi-on-off/com.asksven.betterwifionoff

[42] F. LimitedL, *Google Play Store: FBReader: Favorite Book Reader*, (accessed September, 2020). [Online]. Available: https://play.google.com/store/apps/details?id=org.geometerplus.zlibrary.ui.android&hl=en

[43] N. Pelstin, *geometer/FBReader-Android-2r*, (accessed September 04, 2020). [Online]. Available: https://github.com/geometer/FBReader-Android-2/tree/master/fbreader/app/src/main/java/org/geometerplus/fbreader/fbreader

[44] F. Limited, *Apkpure: FBReader Favorite Book Reader*, (accessed September 04, 2020). [Online]. Available: https://apkpure.com/fbreader-favorite-book-reader/org.geometerplus.zlibrary.ui.android

[45] I. ITO Technologies, *Google Play Store:Pedometer - Step Counter*, (accessed September, 2020). [Online]. Available: https://play.google.com/store/apps/details?id=com.tayu.tau.pedometer&hl=en

[46] j4velin, *Github: j4velin/Pedometer*, (accessed September 04, 2020). [Online]. Available: https://github.com/j4velin/Pedometer

[47] I. ITO TechnologieS, *Apkpure:Pedometer Step Counter 5.33 for Android*, (accessed September 04, 2020). [Online]. Available: https://apkpure.com/pedometer-step-counter/com.tayu.tau.pedometer

[48] X. Li, Y. Yang, Y. Liu, J. P. Gallagher, and K. Wu, "Detecting and diagnosing energy issues for mobile applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 115–127.

Appendix: The XML Code of the Subject Programs

- **XML Code for the Control Flow Graph of the Event E2:**

<graph version="1.0" encoding="UTF-8"

<GraphXML>

<graph version="1.0" vendor="www.drgarbage.com"
id="FindMe">

<node name="2">

<label>41</label>

</node>

<node name="3">

<label>42</label>

</node>

<node name="4">

<label>43</label>

</node>

```
<node name=”5”>
<label>44</label>
</node>
<node name=”6”>
<label>45</label>
</node>
<node name=”7”>
<label>46</label>
</node>
<node name=”8”>
<label>47</label>
</node>
<node name=”9”>
<label>48</label>
</node>
<node name=”10”>
<label>49</label>
</node>
<node name=”1”>
<label>START</label>
```

```
</node>

<node name=”11”>

<label>EXIT</label>

</node>

<edge source=”1” target=”2”>

<label/>

</edge>

<edge source=”2” target=”3”>

<label/>

</edge>

<edge source=”3” target=”4”>

<label/>

</edge>

<edge source=”4” target=”5”>

<label/>

</edge>

<edge source=”5” target=”6”>

<label/>

</edge>

<edge source=”6” target=”7”>
```

```
<label/>
</edge>
<edge source=”7” target=”8”>
<label/>
</edge>
<edge source=”7” target=”9”>
<label/>
</edge>
<edge source=”2” target=”10”>
<label/>
</edge>
<edge source=”4” target=”10”>
<label/>
</edge>
<edge source=”8” target=”10”>
<label/>
</edge>
<edge source=”9” target=”10”>
<label/>
</edge>
```

<edge source=”10” target=”11”>

<label/>

</edge>

<label/>

</edge>

</graph>

</GraphXML>

- **XML Code for the Control Flow Graph of the Event E5:**

<graph version=”1.0” encoding=”UTF-8”

<GraphXML>

<graph version=”1.0” vendor=”www.drgarbage.com”
id=”FindMyLocation”>

<node name=”2”>

<label>75</label>

</node>

<node name=”3”>

<label>76</label>

</node>

<node name=”4”>

<label>77</label>

</node>

```
<node name=”5”>
<label>78</label>
</node>
<node name=”6”>
<label>79</label>
</node>
<node name=”7”>
<label>80</label>
</node>
<node name=”8”>
<label>81</label>
</node>
<node name=”9”>
<label>82</label>
</node>
<node name=”10”>
<label>83</label>
</node>
<node name=”11”>
<label>START</label>
```

```
</node>

<node name="12">

<label>EXIT</label>

</node>

<edge source="11" target="2">

<label/>

</edge>

<edge source="2" target="3">

<label/>

</edge>

<edge source="3" target="4">

<label/>

</edge>

<edge source="4" target="5">

<label/>

</edge>

<edge source="5" target="6">

<label/>

</edge>

<edge source="5" target="7">
```

<label/>

</edge>

<edge source=”6” target=”7”>

<label/>

</edge>

<edge source=”7” target=”8”>

<label/>

</edge>

<edge source=”8” target=”9”>

<label/>

</edge>

<edge source=”3” target=”10”>

<label/>

</edge>

<edge source=”7” target=”10”>

<label/>

</edge>

<edge source=”9” target=”12”>

<label/>

</edge>

```
<edge source=”10” target=”12”>
<label/>
</edge>
<label/>
</edge>
</graph>
</GraphXML>
```

- **XML Code for the Control Flow Graph of the Event E6:**

```
<graph version=”1.0” encoding=”UTF-8”
<GraphXML>
<graph version=”1.0” vendor=”www.drgarbage.com”
id=”SendLocationBySMS”>
<node name=”2”>
<label>65</label>
</node>
<node name=”3”>
<label>66</label>
</node>
<node name=”4”>
<label>67</label>
</node>
```

```
<node name=”5”>
<label>68</label>
</node>
<node name=”6”>
<label>69</label>
</node>
<node name=”7”>
<label>70</label>
</node>
<node name=”8”>
<label>71</label>
</node>
<node name=”9”>
<label>72</label>
</node>
<node name=”10”>
<label>START</label>
</node>
<node name=”11”>
<label>EXIT</label>
```

```
</node>

<edge source=”2” target=”3”>

<label/>

</edge>

<edge source=”3” target=”4”>

<label/>

</edge>

<edge source=”4” target=”5”>

<label/>

</edge>

<edge source=”5” target=”6”>

<label/>

</edge>

<edge source=”6” target=”7”>

<label/>

</edge>

<edge source=”7” target=”8”>

<label/>

</edge>

<edge source=”8” target=”9”>
```

<label/>

</edge>

<edge source="9" target="11">

<label/>

</edge>

<label/>

</edge>

</graph>

</GraphXML>