CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, ISLAMABAD

A Static Analysis Based Approach to Detect Energy Bugs in Android Applications

by

Asia Shahab

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

in the

Faculty of Computing
Department of Computer Science

2020

Copyright © 2020 by Asia Shahab

All rights reserved. No part of this thesis may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, by any information storage and retrieval system without the prior written permission of the author.

This thesis is wholeheartedly devoted to my beloved Parents. I would also like to dedicate this thesis to my sisters Fatima Sultana and Sumia Kalsoom for pushing me to finish thesis on time. I have a special feeling of gratitude for my beloved parents, siblings and friends. Special thanks to my kind supervisor whose uncountable confidence enabled me to reach this milestone.

CERTIFICATE OF APPROVAL

A Static Analysis Based Approach to Detect Energy Bugs in Android Applications

by

Asia Shahab

(MCS181011)

THESIS EXAMINING COMMITTEE

S. No.	Examiner	Name	Organization
(a)	External Examiner	Dr. Yaser Hafeez	UAAR, Rawalpindi
(b)	Internal Examiner	Dr. Masroor Ahmed	CUST, Islamabad
(c)	Supervisor	Dr. Aamer Nadeem	CUST, Islamabad

____

Dr. Aamer Nadeem
Thesis Supervisor
October, 2020

____

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

____

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

Author's Declaration

I, **Asia Shahab** hereby state that my MS thesis titled "**A Static Analysis 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.

(Asia Shahab)

Registration No: MCS181011

Plagiarism Undertaking

I solemnly declare that research work presented in this thesis titled "**A Static Analysis 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.

(Asia Shahab)

Registration No: MCS181011

Acknowledgements

I am thankful to ALLAH for blessing me with strength to complete my thesis. I would like to express my special thanks to my enthusiastic supervisor and soul father **Dr. Aamer Nadeem** for his help and motivation. I thank him wholeheartedly for his tremendous support. I am thankful to Mr. Qamar uz Zaman who helped me in my research. I highly appreciate his patience and motivation. I would also like to thank all members of Center for Software Dependability (CSD) especially Sarmad Hassan for motivating me to complete my thesis on time and for kind help and guidance during my research work.

My deepest gratitude goes to my beloved parents and siblings for tolerating my mood swings. A very special thanks to my father, Nizam ud Din for believing in me. Heartily thanks to my mother, Sakhi Marjana for her love and moral support. Sincere thanks to my sisters Fatima Sultana, Sumia Kalsoom, Hina Shafaq and all family members for their love, prayers and support during my study.

A true friend is one who thinks you are a good egg even if you are half-cracked. I am in great indebtedness to my friends Shahila Sadiq, Ayesha Naseer, Iqra Kiran and Ruquia Zareen for their help and motivation to achieve this goal.

Last but not the least and without hesitation, I would like to thank myself, for surviving all the stress.

(Asia Shahab)

Registration No: MCS181011

Abstract

Smartphone is the most popular device becoming ubiquitous and its usage has experienced significant growth in the recent years because of its rich set of features and services. Due to its rapid evaluation in today's era, it is improving exponentially in terms of computation power and application complexity. It is evident that with the advancement in smartphones features numerous innovative and complex applications are appearing in smartphones day by day. Besides, many applications suffer from serious energy inefficiency problems due to which usually higher amount of power is consumed than expected and battery is drained/ depleted in a short time. Energy inefficiencies appear when applications access the device resources inappropriately (i.e. not releasing Wi-Fi/GPS/Wakelocks).

We aim at energy inefficiencies in smartphones, namely energy bugs, caused by smartphone applications. For the detection of energy bugs several approaches were proposed which are based on static, dynamic and hybrid analysis. Static approaches use the static analysis and can find energy bugs without executing the code. Additionally, static approaches are cost effective but there is an issue of generation of false positives due to not executing the application. On the other hand, dynamic approaches use dynamic analysis therefore applications are executed with several numbers of test cases. However, due to the execution of large number of paths or test cases dynamic approaches become more costly and more time consuming. Moreover, hybrid approaches use both static and dynamic analysis therefore these approaches are also costly due to the execution of test cases.

This study focuses on static analysis approaches to overcome the false positives. Therefore, we proposed a technique to investigate the false positives to effectively detect the energy bugs in Android applications. First of all, we have applied the existing static analysis based approach and different bug prone paths are identified. After that, all the identified bug prone paths are further analysed. Besides, by using the symbolic execution path conditions are created for each buggy path that

are further evaluated on the constraint solver to investigate that either identified buggy paths are indicting real energy bugs or false positives.

We tested our approach towards a number of open-source real-world applications and analyze to demonstrate the efficiency and accuracy of our approach by identifying the infeasible paths. Our experimental results show that we identified most of the real energy bugs as well as false positives. This study helps application developers to identify where and how the abnormal battery drains issues have occurred. Further, we discussed future work regarding this research.

Contents

Author's Declaration iv

Plagiarism Undertaking v

Acknowledgements vi

Abstract vii

List of Figures xi

List of Tables xii

Abbreviations xiii

1 Introduction 1
- 1.1 Energy Bugs 2
- 1.2 Detection of Energy Bugs 4
- 1.3 State-of-the-Art Approaches 7
  - 1.3.1 Static Approaches 7
  - 1.3.2 Dynamic Approaches 8
  - 1.3.3 Hybrid Approaches 8
- 1.4 Problem Statement 8
- 1.5 Scope 9
- 1.6 Research Questions 9
- 1.7 Research Methodology 10
- 1.8 Research Contribution 11
- 1.9 Thesis Organization 12

2 Literature Review 13
- 2.1 Energy Bugs Detection Techniques 14
  - 2.1.1 Hybrid Energy Bugs Detection Techniques 15
  - 2.1.2 Dynamic Energy Bugs Detection Techniques 16
  - 2.1.3 Static Energy Bugs Detection Techniques 20
- 2.2 Analysis and Comparison 24

3 Proposed Approach **34**
3.1 Control Flow Graph Generation 36
3.2 State Taint Analysis 38
3.3 Symbolic Execution 41
3.4 Constraint Solver 44

4 Implementation **46**
4.1 Control Flow Graph Generation 46
4.2 State Taint Analysis 49
4.3 Symbolic Execution 51
4.4 Constraint Solver 52

5 Results and Discussion **54**
5.1 Android Applications 55
5.2 Comparison 71
5.3 Mapping Study 75

6 Conclusion and Future Work **77**
6.1 Future Work 80

Bibliography **81**

Appendix **88**

List of Figures

3.1 FlowChart of Proposed Approach 35
3.2 An example code 36
3.3 CFG of example code 37
3.4 Abstract FSA for General Resources 39
3.5 Specific FSA for Wakelock 39
3.6 Path Conditions for example code 43

4.1 An example code 47
4.2 CFG of example code 48
4.3 MiniZinc code and Output of buggy path (TTF) 52
4.4 MiniZinc code and Output of buggy path (TFT) 53
4.5 MiniZinc code and Output of buggy path (TFF) 53

5.1 Source code of Adbwireless Application 55
5.2 CFG of Adbwireless Application 56
5.3 MiniZinc code and Output of buggy path (FTFT) 57
5.4 MiniZinc code and Output of buggy path (FFFF) 57
5.5 Source code of Smspopup Application 58
5.6 CFG of Smspopup Application 59
5.7 MiniZinc code and Output of buggy path (TTF) 60
5.8 MiniZinc code and Output of buggy path (TFT) 61
5.9 MiniZinc code and Output of buggy path (TFF) 61
5.10 Source code of Fbreader Application 62
5.11 CFG of Fbreader Application 63
5.12 MiniZinc code and Output of buggy path (TTT) 64
5.13 MiniZinc code and Output of buggy path (TFT) 65
5.14 MiniZinc code and Output of buggy path (TTFF) 65
5.15 Results of Selected Applications 71
5.16 Comparison of Total Energy Bugs 73

List of Tables

1.1 Classification of Energy Bugs 3
1.2 Situations of Energy Bugs 6

2.1 Comparison of Static Techniques 25
2.2 Analysis of Static Techniques with different aspects 31

3.1 Graph Coverage Criteria's 41
3.2 Test paths, Buggy paths and Resource details 41
3.3 Path conditions and Results of buggy paths 45

4.1 Graph Coverage Criteria's 49
4.2 Test paths, Buggy paths and Resource details 50
4.3 Path Conditions for buggy paths 51

5.1 Test paths, Buggy paths and Resource details 56
5.2 Path Conditions for buggy paths 57
5.3 Test paths, Buggy paths and Resource details 60
5.4 Path Conditions for buggy paths 60
5.5 Test paths, Buggy paths and Resource details 64
5.6 Path Conditions for buggy paths 64
5.7 Results for Selected Applications 67
5.8 Comparison of Selected Applications 72
5.9 Mapping Study 75

Abbreviations

API	Application Program Interface
APK	Android Package
CFG	Control Flow Graph
EFG	Event Flow Graph
FNs	False Negatives
FPs	False Positives
FSA	Finite State Automata
HTTP	Hypertext Transfer Protocol
OS	Operating Systems
TP	Test Path
TPs	True Positives
TR	Test Requirement

Chapter 1

Introduction

Smartphones are becoming popular devices in marketplace and their use is increasing with the passage of time [1]. Faruki et al.[2] stated that due to cost effectiveness, internet, games, location based services, messages and multimedia services smartphone has become ubiquitous. In order to meet the customer's demands, many I/O components are added in modern smartphones [3]. These components include GPS, Wi-Fi, camera and many more. Despite of converting these smartphones in modernized form, issue of high energy consumption remains a major factor [4]. On the other hand, mobile phones have limited battery timings and limited battery power [5].

In short, mobile phones require frequent battery recharge. Hence, it can be said that one of the most critical resources in smartphone is battery life and also the battery development in mobile phones is slower as compared to the hardware advancements. Despite of all these factors, smartphone users like to spend most of their time on using mobile phones [5]. With the passage of time, mobile phone companies introduced new features along with improved processing power and battery capacity but the increase in battery capacity was not that much as compared to processing power. A study proved this fact by comparing Nokia 9000 Communicator and Samsung S3 [6]. These researchers further reported that from 1996 to 2012 processing power was increased from 24MHz to 1.4GHz. However, the battery capacity was only improved from 800mAH to 2100mAH.

It is evident that with the advancement of the smartphones specifications many innovative and complex applications are appearing in smartphones on daily basis. But the quality of these advanced applications and efficiency of new codes cannot be guaranteed. The reason is that every developer has his/her own expertise and has different level of skills that varies from novice to expert. Also, most of the applications need more energy to perform their functions. These energy hungry applications lead towards short battery time and more power consumption which ultimately results in unsatisfied user [6]. Researchers stated that user's satisfaction can be increased by improving the energy efficiency of mobile applications [7].

1.1 Energy Bugs

Pathak et al. [8] talked about abnormal battery drain issues that are caused by different reasons including hardware, software and external. These researchers outlined battery drained issue and performed the first study to present the taxonomy of the energy bugs. They also reported the percentage of battery drain caused by hardware and software i.e. 22.93% and 35.10% respectively. Xiao Ma et al. [9] also observed that most of the drainage of battery caused by application-related bugs. These bugs cause user frustration and the severity of these bugs can be estimated by the reports that are present on the internet forums.

Banerjee et al. [6] stated about the energy inefficiencies in Android applications. Energy-inefficiencies can be categorized into two types i.e. energy bugs and energy hotspots. Serious energy issues arise in smartphones due to energy bugs. Energy bug, as name suggests, causes an error to the energy of the system or reduce the battery life in smartphone [8]. In case, when there is no user activity or when applications are not used (when applications are terminated or closed), there should be low energy consumption. But in fact, these applications consume high power and more energy even if there is no user activity and applications are terminated even after it has completed execution. High power consumption indicates the existence of an energy bug in an application [10]. While a scenario in which an application

is executing on a smartphone and it starts consuming abnormally high amount of battery power despite low hardware resource utilization execution is outlined in the study as energy hotspots. Banerjee et al. [6] proposed the classification of energy bugs that can be found in Android applications. This classification shows the different natures of smartphone energy bugs. Table 1.1 represents the classification of the energy bugs that can be present in an Android application. As

Table 1.1: Classification of Energy Bugs

No.	Category	Energy Bugs
1.	Hardware Resource	**Resource Leak:** Existence of resource leak bug indicates that before exiting the requested/acquired resources (such as Wi-Fi) are not released.
2.	Sleep-State Transition Heuristics	**Wakelock Bug:** If Wakelocks are used in an improper way, it can result in high-power consumption even if application has finished its execution [8].
3.	Vacuous Background Services	**Vacuous Background Services:** When Applications mishandles background services but doesn't remove the service explicitly before exiting, the service keeps on reporting data even though no application needs it. This situation leads to high energy consumption.
4.	Defective Functionality	**Immortality Bug:** A situation where a buggy application drains battery, upon being closed by the user, respawns, enters the same buggy state and continues to drain battery [8].

shown in Table 1.1, if resource leak bug exists, it indicates that before exiting, the requested resources are not released. Sometimes, when there is need to awake an application, a power management mechanism is used to keep it awake. For that purpose, wakelocks are acquired. But if wakelocks are not properly used by application then device will be stuck in high power state although application has completed its execution. Wakelock bug was detected in many mobile applications such as Gallery app, Google Calendar, email apps, weather apps. Vacuous background services like updates about location or sensors should be perfectly handled by applications. But if these services are mishandled by application and service is not removed explicitly before exiting then service will continue reporting data

even though no application needs it. In such case, vacuous background services bug occurs. This bug was observed in different applications such as GPS (Map based Apps), location updates, sensor updates. Defective Functionality can be defined as a scenario where user closes an application but still that application drains battery power, re-spawns, and enters the same buggy state, an immortality bug occurs. This bug was detected in different applications such as Media Server and Google Maps.

1.2 Detection of Energy Bugs

Android is an open-source operating system (OS) designed for mobile devices such as smartphones. We choose Android as our target platform primarily due to its relevance in the real world. According to study [2] OS of Android smartphone has captured more than 75% of the total market-share. Now, they are being used to perform daily activities. However, some of the applications in smartphones are energy hungry which results in short battery timing and high power consumption [11]. This can also results in unsatisfied consumers [6]. It is also revealed that smartphones have low battery power and they need frequent recharge. Even then, users preferably use smartphones as long as possible [5]. High energy consumption caused by an application is because of existence of energy bug. This energy is consumed even if application does not work or it is stopped. The researchers reported that energy bugs can be introduced due to the mistakes of the developers during coding [12]. From these studies, it can be analyzed that developers of energy inefficient applications face challenges of energy and receive many complaints from the users.

Pathak et al. [13] studied major components of power consuming. These components include I/O components like CPU, Wi-Fi, radio, Camera and GPS. In order to access these components in an application, a set of system calls (APIs) provided by the Android SDK framework is used. Similarly, [6] analyzed that except CPU, other I/O components like Wi-Fi, radio, Camera and GPS can only be accessed

via a set of system calls (APIs) provided by the Android SDK framework. Additionally, a set of system calls (APIs) is also used to access background services like updates about location or sensors, the power management functionality and some other hardware sources.

According to Banerjee et al. [6], most of the energy bugs are exposed through the invocation of system call(s). The basic reason of energy inefficiencies is when the application accesses the device resources in an inappropriate manner (e.g. not releasing Wi-Fi or GPS, Wakelocks or expensive sensor updates). This ultimately results in short battery timings and battery life. Therefore, to build energy-efficient applications, it is crucial for the developer to know these energy inefficiencies in the application code.

For example, one of the most popular Android application "Osmdroid" contains energy inefficiency problem i.e. Osmdroid Issue 53 [14]. This problem was also reported by users of this particular application. This issue was raised by switching between different activities, for example, switching from MapActivity to any other activity with disabled location tracking, this keeps acquiring GPS data to extract an invisible map due to an unreleased wakelock. Resultantly, useless location sensing reduces battery energy.

The resource in an application can be acquired as: (i) local resource and (ii) global resource. Local resource is declared in a function while global one is declared outside the functions and within a class. Also, developer releases local resource while function ends whereas global resource is released on all exit paths from its request point [15].

Moreover, researchers reported that for program correctness it is vital to verify that program is using the resources effectively or not [16]. Improper usage of the resources cause short battery life (also known as energy leak problem) as it causes the system to run out of resources. If the resources are unnecessarily opened or an obtained resource is closed without any usage then it will result in shorter battery life.

Furthermore, actual flow of source code can be complex and there can be several nodes and edges in the CFG of an application. Besides, for the purpose of energy bugs detection, we should focus on those paths of programs/source code on which resources are acquired but never released or released without being used as these paths contain energy bugs. Inappropriate resource usage leads to resource starvation or too much short battery life. In other words, to overcome the battery leakage we have to detect those paths on which energy bugs can arise or we have to detect those paths which contain energy bugs. So, different graph coverage criteria's can be applied i.e. node coverage, edge coverage, edge-pair coverage or loop coverage etc. We have applied the edge coverage criterion on the control flow graph. As one resource can leads to different states from different paths, therefore, it is necessary to cover all the edges of the graph. For this purpose we have used edge coverage criterion and it was able to cover all the edges of the graph. However, more stronger coverage criteria's than edge coverage criteria can be applied but these criteria's will increased the cost due to generation of more number of paths as well as effectiveness will not be much increased. Moreover, in 2014, Li et al. [17] reported that 75% of the application spends more than 89% of their network energy in HTTP(Hypertext Transfer Protocol). This designates that some resources are more energy hungry. Therefore, incorrect usage will lead to energy leak problem i.e. HTTP requests making is the most energy consuming network operation. The current study focuses on the energy bugs detection in Android applications due to resource leakage. Table 1.2 signifies particular situations of energy bugs that are deliberated in this study.

Table 1.2: Situations of Energy Bugs

No.	Category	Energy Bug
1.	Open (O)	**O:** A situation where the acquired resource is neither used nor released.
2.	Unclosed-ness (O and U)	**U:** This situation can occur if the acquired resource is used, but not closed.
3.	Unused-ness (O and C)	**C:** It happens when the resource is opened and closed without any use.

As shown in Table 1.2, the resource that is acquired must be used before closing or should be released on time. Most of the energy bugs arise because of unclosed-ness (i.e., O and U). The reason behind is that some of the programmers do not remember to close the resource when they exit. Such as, if the resource named "Wi-Fi" is acquired and used but not closed at the end then it will consume more energy. Other reason is unused-ness (i.e., O and C). This situation occurs when the programmer opened the resource in advance and closed at the last minute without considering it is in need or not. In other words, we can explain it as the resource is acquired and closed without examine that whether it is used or not. Such as, if the resource named "Wi-Fi" is acquired and closed without performing any task then it will consume more energy. Moreover, the situation of (O) can occur if unnecessary resources are acquired by programmers.

1.3 State-of-the-Art Approaches

Several static, dynamic and hybrid analysis based approaches to detect the energy bugs in Android applications exist in the literature but most of the studies focus on the second situation of energy bugs named as "unclosed-ness".

1.3.1 Static Approaches

Static approaches perform static analysis of code to detect the energy bugs without executing the code of the application and can find energy bugs. Additionally, static analyses are valuable due to the ability of examining all program paths [18]. However, static analyses are not able to evaluate the conditions of source code and there may be paths that are infeasible (invalid paths) which designate that these paths will fail to execute [19]. Researches stated that static analysis may generate false warnings or may miss important bugs while testing [20]. Consequently, these approaches generate more number of false positives because code is not executed and infeasible paths lead to results in false positive. Therefore, for better detection

of energy bugs we need to reduce the number of false positives. In order to overcome this issue this study aims to eliminate the false positives by deliberating the path conditions which help to identify the false positives (FPs) reported from the existing static approach proposed by Xu et al. [21].

1.3.2 Dynamic Approaches

With static analysis, code is not executed so it is not confirmed that bug will actually arise or not. Whereas, dynamic analysis executes the code so possibilities of bug detection are better than static analysis. In other words, for accurate detection of energy bugs dynamic analysis approaches are more effective. However, test cases are executed which makes dynamic analysis approaches more costly. Additionally, due to the execution of large number of paths these approaches become more costly.

1.3.3 Hybrid Approaches

Hybrid approaches are the combination of both static and dynamic analyses. Due to providing the advantages of both static and dynamic analyses, hybrid approaches are valuable. As, these approaches are able to provide the advantages of both static and dynamic analyses however, due to dynamic analysis its execution time and cost can also be increased.

1.4 Problem Statement

The current state-of-the-art demarcates that the researchers have proposed various techniques to detect the energy bugs where most of the studies are based on second situation of energy bugs named “unclosed-ness” but very few studies are based on open and unused-ness. Moreover, some studies are based on static analysis and some of them are based on dynamic analysis. Dynamic analysis approaches are

more costly due to the execution of test cases. Conversely, static analysis analyse the code without execution. Therefore, existing static analysis approaches generate more number of false positives. Besides, these approaches consider all the paths for the detection of energy bugs including feasible and infeasible. If there is a bug in infeasible path it will also be consider as bug from static analysis approaches although it is not a bug but false positive.

1.5 Scope

Various approaches are used to detect different type of bugs and hotspots in Android applications. Serious energy issues arise due to energy bugs. Therefore, the scope of this study is to identify the energy bugs in Android applications and to consider two types of energy bugs including resource leak bugs and wakelock bugs. Other type of bugs in program such as logical bugs, performance bugs and hotspots are not considered. Moreover, we focused on the static analysis approaches to eliminate the false positives (FPs). The reason for the selection of static analysis approaches is that these approaches generate more number of false positives.

1.6 Research Questions

Based on the objectives of the study, following research questions are designed:

RQ 1: What are the shortcomings of existing static energy bugs detection techniques?

Through in depth study of literature, static analysis based techniques implemented in the field of energy bugs are investigated thoroughly, and their gaps are identified.

RQ 2: How to overcome the shortcomings of existing static energy bugs detection techniques?

False positives are generated due to the static analysis and for the identification of FPs a static analysis based approach is proposed. Path conditions are created for each erroneous path by using symbolic execution and constraint solver is used to determine the infeasible paths or FPs.

RQ 3: Does the proposed approach overcome the false positives when compared with existing techniques?

Several experiments are performed on different Android applications for determining the effectiveness of the proposed technique over existing static analysis techniques.

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

1.7 Research Methodology

1. First of all, we have done literature review to identify the most relevant hybrid, dynamic and static energy bugs detection techniques. After studying numerous techniques, we have reached the conclusion that due to static analysis, static analysis approaches produce false positives.
2. To overcome the gap in existing static analysis techniques, we have proposed a solution of adding the path conditions that helps to identify the infeasible paths.
3. Implementation of our approach is performed in the following steps:
    (a) In the initial phase, we collect the real world Android applications and read their source codes for the exploration of energy bugs.
    (b) In the next step, Control flow Graph is created from each source code of application.
    (c) We have applied the State taint analysis to detect the energy bugs.

   i. Control flow Graph, graph Coverage criteria is taken as input and resource usage protocols (Automata) is used as a guide.

   ii. CFG is traversed according to the Edge coverage criterion and test paths are created.

   iii. After that, it is checked that whether resource action obeys the corresponding resource protocol (resource automata) or not; until all the paths are checked. Those paths are considered as erroneous/buggy on which an action sequence does not satisfy the resource protocol.

   (d) After that, by using Symbolic execution path conditions are created for each erroneous path.

   (e) Paths conditions are evaluated by constraint solver "MiniZinc" and infeasible paths which lead to FPs are identified.

4. For comparison, false positives are used as the main parameter. We have also compared our approach with existing approach [21] by using their data set for the purpose of evaluation.

1.8 Research Contribution

1. In this research work, we have proposed the static approach for better energy bugs detection in Android applications. The proposed technique is comprehensively evaluating control flow graph (CFG) and resource usage protocol (automata). The graph coverage criterion named "Edge coverage criteria" is applied on the CFG, and a resource usage protocol is used as a guide to identify the buggy paths in a CFG. Paths which are indicating energy bugs (buggy paths) are identified.

2. Symbolic execution is used for creation of path conditions only for identified buggy paths to overcome the complexity of generated constraints.

3. The main identified problem of static analysis approaches is that these approaches generate more number of false positives so we have used false positives as the main parameter to identify that either we are able to overcome the limitations of existing studies or not. To overcome the false positives or to identify the infeasible paths which leads to false positives, path conditions are evaluated by using the constraint solver also known as query solver.

4. Although our approach is static approach but false positives that are generated due to static analysis are reduced. Therefore, our approach is more scalable and helps in reducing the overall analysis time although with the static analysis.

5. To evaluate the effectiveness of our approach, we performed experiments on different 13 real world Android applications, which are downloaded from GitHub and Google code. In out experimental results, our approach reported the real energy bugs as well as removes the false positives in 5 of the tested applications.

1.9 Thesis Organization

The rest of study is organized as follows:
Chapter 2 surveys the existing work on energy bugs detection. Chapter 3 discusses the proposed approach and its details. Chapter 4 discusses the implementation details. Chapter 5 presents the experimental results of proposed approach and comparing it with existing techniques to evaluate the performance of our proposed technique followed by Chapter 6 which finishes up the entire work and furthermore gives some future research directions.

Chapter 2

Literature Review

Chapter 1 provides sufficient details on scenarios that guide us to define the problem statement. This chapter focuses on critical analysis of all the state-of-the-art approaches, as every research study is dependent on the previous study, that have already been performed in this field. The number of new ideas for detection of energy bugs in Android applications have been proposed as the numbers of Android applications are increasing day by day and incorporating features like multiple cores, GPS location tracking and large screens etc.

Energy inefficiencies in smartphone applications can be identified by a class of bugs which is known as energy bugs. Pathak et al. [8] studies the class of bugs on smartphones. The focus of their study was on the challenges that are associated with the dealing the energy bugs. As per their definition the energy bug is an error that results in unexpected high energy consumption in the system. The energy consumption can be associated with OS, hardware, firmware or external. Resultantly, the mobile device having an issue/error runs out of power. Furthermore, according to their results, application related energy bugs were the most noticeable type of energy bugs. They also identified the different reasons of battery drain caused by hardware (22.93%) and software (35.10%).

To avoid energy bug issues such as short battery life and low energy power, modern smartphones are designed by using different power levels so that battery can be

used for longer time period. In case, if there is an application having energy bug issue will result in inappropriate power state, i.e. even when there is no user activity and power state is still in non-idle state. Most of the energy problems arise when application is unable to access the system resources appropriately, such as not releasing Wi-Fi, or not releasing wakelocks etc [6].

Pathak et al. [13] studied about power consuming components and identified the primary sources of energy consumption in smartphones. These components include I/O components such as Wi-Fi, CPU and GPS. In current smartphones Operating Systems (OSs), power management functionality (e.g. Wakelocks) is used that all the components including CPU will be in the idle state until the application itself instructs the OS to keep it awake. Other types of energy bugs, such as sleep-state transitions and background services can also be found in Android applications [6]. Furthermore, according to researchers, these types of bugs were found in different real life Android applications i.e. Facebook, Location listener and apps that were using GPS like Osmdroid, GPSLogger etc.

In this chapter, different existing energy bugs detection techniques are discussed.

2.1 Energy Bugs Detection Techniques

Energy impact of software is considered vital and cannot be ignored because of higher demand of energy. Additionally, literature is rich about energy inefficiencies detection in the Android applications. Several static, dynamic and hybrid approaches exist for the detection of energy bugs. Hybrid approaches are the combination of both static and dynamic analyses. These approaches are useful because it provides the advantages of both static and dynamic approaches. However, due to dynamic analyses its execution time can be increased and cost can also be increased. Moreover, static approaches perform static analysis of the code to check bugs without executing the code of application. Dynamic approaches analyze the dynamic behavior of code by executing it. Although, due to the execution of code, detection of energy bugs with dynamic analysis is more effective

than static analysis. But there are large number of paths and all paths can never be executed and if so these approaches become more costly. On the other hand, if energy bugs are indicated in any path from static analysis but that path can never be executed then identified energy bug can never arises and will be considered as false positive. Therefore, main focus of this study is on static analysis approach so that we can identify infeasible paths.

It is evident that energy bugs detection has been studied by number of studies but there are few limitations. Following are the different energy bugs detection techniques based on hybrid, dynamic and static analyses respectively.

2.1.1 Hybrid Energy Bugs Detection Techniques

Such as, in 2014, Banerjee et al. [6] presented the hybrid approach, an automatic test generation framework for the detection of energy hotspots and energy bugs in Android applications. The proposed framework consists of two major components, i.e. guidance and hotspot or energy bug detection component. To reveal energy hotspots or energy bugs, they explored the event traces in guidance component. In addition, in hotspot or energy bug detection component, they executed an application on a smartphone and a power meter was attached with it to measures the power consumption of the application. But, still there are chances that portions of code will be unanalysed due to incomplete test generation method. Therefore, it can generate false negatives. Additionally, the exploration algorithm generates events traces by walking through the EFG (Event Flow Graph). But for those UI screen which need inputs through an input-container (such as text-fields) random data were used. Resultantly, the exploration algorithm may not have been able to explore all feasible paths inside an event-handler.

In 2018, Banerjee et al. [15] enhanced previous approach [6] and developed the EnergyPatch, a framework that uses hybrid analysis (i.e. a combination of static and dynamic analysis techniques) for the detection, validation and repair energy

bugs in Android applications. In their prior approach [6], these researchers generated the EFG of user-generated events. Whereas, in enhanced approach, CFG (Control Flow Graph) of the event-handlers along with each event of the EFG was generated. Each event of an EFG was associated with CFG. Furthermore, Energy bugs were detected by using abstract interpretation and after that for validation repair expressions were attached. But still there are chances of incorrect results due to incomplete EFG generation, different events such as context events were not considered. Additionally, CFG was generated for EFG, but no graph coverage criterion was applied so it was generating overestimated results, including infeasible paths and false positives.

In 2020, Li et al. [22] considered previously unaddressed energy issues in [6] due to unnecessary workload on hardware components i.e. extremely frequent operations, CPU and GPS. Additionally, main root causes of energy related issues were also revealed i.e. unnecessary workload and excessively frequent operations. But limitation was that the source codes of apps were statically analysed and due to which infeasible paths were generated that lead to false positives.

2.1.2 Dynamic Energy Bugs Detection Techniques

In 2012, Zhang et al. [23] developed a tool named "Automatic Detector of Energy Leaks (ADEL)" for detecting energy leaks in network communication for mobile applications such as Wi-Fi and 3G interfaces. This tool consists of dynamic taint-tracking in which each data object is labeled with a tag and the network data is also tracked from its origination to its use or until its deletion. Additionally, it helps to identify energy leaks resulting due to network download or unnecessary network communications. Main focus on network downloads is because of two main reasons. First, network devices, for example, cellular devices are more energy hungry. Almost 70% of energy is consumed due to downloads among the network transmissions. Second, in network communication major amount of energy is wasted. However, there were few limitations of this study. First of all, ADEL ignores the control flow and tracks the data flow. Due to the negligence of control

flow it was generating false negatives. Secondly, false positives were generated during taint propagation i.e. in "Busmap" application, text information for one bus stop was contained in each data units, and in the same packet 2 to 3 data units were packed. Resultantly, information for unused bus stops will be identified wrongly if any bus stop in the packet was used.

In 2013, to handle the misuse of wakelocks Kim et.al [24] presented a technique "wakescope". Wakelock acquisition and release behaviour was tracked with the help of wakescope. It was able to detect the improper handling of a wakelock to avoid the energy waste in a smartphone device and it was also able to notify the misuse of a wakelock handling. Inadequacy in this study was that wakescope was not able to detect the problem at the source-level due to which this work was not able to identify the cause of the problem in detail so false negatives can be generated.

In 2015, Abbasi et al. [10] studied and investigated about operational definition of energy bugs. To detect the presence of energy bugs, operational definition can be easily translated into a procedure. Moreover, these researchers investigated the existence of energy bugs when different applications are updated. Proposed definition was validated with measurements and realistic energy bug examples. However, this study also includes few drawbacks. It can be said that framework was not fully implemented for detecting the targeted types of energy bugs is a limitation of this particular study. Furthermore, executed test cases did not have features or functionalities that can be added to upgraded application which will leads to results in false negatives.

In 2015, Ferrari et al. [25] presented the design and implementation of a Portable Open Source Energy Monitor (POEM) to permit the developers to automatically test and measure the energy consumption of the call graph, the basic blocks, and the Android API calls. Furthermore, the developers were able to locate energy leaks with high accuracy. But, the runtime and energy consumption were increased because when method is run for logging purpose POEM needs to access the file system, which will results in a deterministic logging cost. Additionally, it was

unable to capture those events which can take place so it will results in false negatives.

In 2018, Abbasi et al. [26] discussed about the concept of Application Tail Energy Bugs (ATEBs), its formal definition and identified root causes of tail energy bugs. The researchers also developed the tool ATEBs detector for the detecting the presence of ATEBs. However, there were no information identified that how to release the wakelocks automatically when they have been acquired but not released at the end even when they were not needed. Moreover, the tool was not fully implemented to support other types of application components i.e. listeners, audio, and wireless services. Therefore, false negatives can be generated due to neglecting other types of application components.

In 2013, an automated approach was presented by Liu et al. [14] and their main focus was on an application's sensory data utilization at different states. Additionally, energy inefficiency problems were reported and their root causes were identified. Moreover, Android specifications were used to derive the application execution model (AEM) to describe the relation between the event handlers for locating the misuse of sensor listeners i.e. which sensor listeners have forgot to unregister at the end of execution. Furthermore, researchers implemented the GreenDroid tool. But, some limitations of this study were still there. For example, all the event handlers were not able to schedule by one state machine. Because of this, it was difficult to manage. Real world Android applications contain hundreds of event handlers but calling relationships between event handlers were manually specifying by the researchers which was not practical. Likewise, complex user inputs (i.e. password) were unable to be generated because of randomly generation of mock sensory data. Due to randomly generation false negatives were produced. Moreover, unknown type of energy waste i.e. wakelock was not diagnosed which causes severe energy waste. As well as, many new features of newer versions of Android were not supported because it was concerning Android 2.3.

In 2014, Liu et al. [27] enhanced previous approach [14] by checking the behaviour of activation and deactivation of both wakelocks and sensors. To identify whether

the application was using sensory data in an appropriate manner or not, the sensory data tracking was done. However, high code coverage was not possible due to complex inputs i.e. user gestures or video inputs were unable to generate, which was results in false negatives. Moreover, event handlers were unable to schedule in the same way as real executions did, due to not considering dynamic GUI updates at the time of implementation of GreenDroid. Additionally, GreenDroid generates sensory data in random manner and feeds them to a running app for sensory data utilization analysis. But some states that were only reachable by some specific sensory data inputs were not reachable by GreenDroid. It was also considered in GreenDroid that an app transfers to a new state after it has finished handling an event i.e. click button, without considering that which program path is executed during this event handling. However, different execution paths can lead to different app states, which were not handled in GreenDroid.

In 2016, Original GreenDroid [14] was further extended by Wang et al. [28] and they re-implemented the GreenDroid on the newest version of Java Path finder (JPF) which gave the ability to supports new features of Android. Another feature was also added by researchers that E-GreenDroid was able to form one state machine to schedule all the event handlers. Furthermore, it was providing a state machine based on AEM, which was reusable for any Android app analysis. However, in this study, AEM was vague so simulation of runtime behaviours of services was killed, due to which false negatives were generated. Further, they were immediately checking for sensor listeners misuse even when activities have not finished their lifecycles yet, which was causing false positives. Moreover, some patterns of wakelock misuses i.e. multiple wakelocks acquisition were neglected therefore, false negatives can be generated.

In 2017, Li et al. [29] implemented the enhanced approach of GreenDroid [27] as a tool CyanDroid. The approach was generating sensory data using multidimensional white-box sampling to explore different app states, tracking their propagation and analyzing their utilization. Additionally, a feature of considering the execution of program paths during event handling was also added by researchers that were not entertained in previous work done by Liu et al. [27]. The drawback

was that only location-aware Android applications were concerned. Moreover, complicated user events (i.e. screen unlock event, valid user name and password) were not simulated when text inputs was needed by Android application, random strings were generated, so some application states were not reachable therefore generating false negatives.

In 2017, another extension was done by Liu et al. [30] and NavyDroid was developed. More energy inefficiency bugs in Android application were discovered than existing work E-GreenDroid did. Along with the GreenDroid previous version functionalities this work was also focusing on multiple patterns of wakelock misuses. Therefore, more complex energy bugs were detected that were caused due to wakelock misuses. Inadequacy in this study was that tool was implemented on Java path finder (JPF) instead of Android virtual machine (AVM) but it may not be consistent with real world Android applications due to only simulate the execution of Android apps.

In 2019, ZHU et al. [31] discussed about the technique that models the power consumption by using system call and seven machine learning models were trained for energy bugs detection. Furthermore, energy bug is not identified in the code, but this procedure identifies commit that introduces energy bugs in the revision history so that developers could find the buggy commit efficiently. But limitation was that training data was not including all the data patterns of test data so, bias and variance will exist.

2.1.3 Static Energy Bugs Detection Techniques

In 2012, Pathak et al. [32] presented a study of energy bugs in Android applications (i.e. no-sleep energy bugs and wakelock bugs). For the purpose of wakelock bugs detection a solution was used which was based on dataflow analysis. To automatically achieve possible no-sleep bugs in an Android application, no-sleep bug detector was developed. Additionally, major causes of no-sleep bugs were also revealed. Inadequacy in this study is that an edge was placed from each run

time exception but that was not handled within routine to EXIT node for that routine. It was creating a path for a lock acquiring definition to reach exit and was generating false positives. Moreover, there was no mechanism to automatically prune out FPs.

In 2013, Guo et al. [33] presented a precise and lightweight static analysis tool Relda that was checking for resource leaks in event handlers. It was able to automatically analyze an application's resource operations and to detect the missing release operations. It was constructing Function Call Graph (FCG) of an application and resource leak summary and report was generated. However, the proposed approach was not context sensitive. For instance, if resource leak is considered and there exist only one releasing resource path, no resource leak report will be given, causing false negatives. Or in other words, due to not considering control flow information to several resource leaks were missed, causing FNs. Other limitation still exist in this study that application can release wakelocks at numerous program points but this was not handled in proposed approach due to the assumption that if wakelocks once acquired should be released at pre-defined set of event handlers but application can release wakelocks at various program points. Therefore, false positives were generated due to vague assumption.

In 2016, Wu et al. [34] further extended Relda presented by Guo et al.[33] and named it as Relda2. It was used to identify resource leak problems in Android application due to different types of resources such as exclusive resources (Camera), memory consuming resources (Media Player) and energy consuming resources (Sensors); it can analyze resource operation of applications and locate the resource leak. Moreover, byte code can be easily modifiable by attackers to change the behaviour of application so the target was byte-code instead of source code. But the study was not free from limitation. One of the major limitations of this study was that false positives were generated due to static analysis.

In 2016, Liu et al. [35] designed a static analysis technique named "Elite" to identify wakelock leakage. Moreover, it makes no assumption on wakelock acquiring or releasing points, but it automatically reasons about need of using wakelocks

at different program points via dataflow analysis. Additionally, eight common patterns of wakelock misuses were identified. However, only two patterns of wakelock misuses were considered. Furthermore, vague call graphs results in incorrect findings and false positives.

In 2016, Wu et al. [36] presented an approach to statically detect energy related defects in Android applications. The aim was to detect the behaviour of missing deactivation in the user interface thread of the Android application. Therefore, two different patterns related to run time energy drain were defined in which location listeners were leaking. However, due to some limitations of static analysis false positives were caused.

In 2016, Kim et al. [37] proposed and evaluated static analysis technique to detect GPU related energy bugs. Three specific types of energy bugs were identified in graphics-intensive applications. Furthermore, they also focused on eliminating drawing commands for producing energy efficient applications and evaluated a method for enhancing energy efficiency of graphics-intensive apps, which focuses on GPU operations. However, two out of three analyses were producing FPs that must be manually eliminated by user.

In 2016, Xu et al. [16] proposed a state-taint analysis for the detection of resource leaks issues. Proposed analysis was implemented as a prototype tool named "statedroid". Furthermore, the researchers taken the open but not used resources problem i.e. if HTTP (Hypertext Transfer Protocol) connection is established it must be used before closing. Additionally, according to the API documentation of resources, appropriate usages of resources were specified as resource protocols for the guidance of resource bug detection. Moreover, the analysis was general because it was able to work with different resource usage protocols. Resource usage protocol was also proposed and it was enhanced with those inappropriate behaviours which may cause energy leaks. The enhanced protocol was used to guide the analysis for energy leak detection in smartphone applications. However, some limitations were still there due to not checking null pointer exception false positives were generated.

In 2017, Jiang et al. [38] developed a technique named Static Application Analysis Detector (SAAD) that can automatically detect energy bugs and analyzed resource leak at background programs. When the application does not release its acquired resources, the resource leak bug occurs. To address this resource leak issue, researchers performed a components call relationship analysis. Furthermore, effective paths were also analysed, those paths in which resource acquisition and release were made considered as effective paths. Additionally,APK (Android Package) file was taken as input and report of resource leaks were generated as output in the proposed framework. However, this approach was producing false positives due to updates to program data i.e. timer and scores, and those false positives must be eliminated by user.

In 2017, Xu et al. [21], extended their prior work [16] in 2017, the researchers used the refined protocols to guide the analysis for energy leak detection. Moreover, more experiments were done on several real Android applications and datasets were also taken from Relda [33] and GreenDroid [27]. However, it was generating false positives and different nodes of control flow graphs can lead to different paths for the same resources, but path conditions were not considered and hence results in false positives.

In 2018, Hall et al. [39] presented an approach for detecting no-sleep bugs using reference counters in consideration of race conditions based on the reaching definitions (RD) dataflow analysis in Pathak et al.'s work [8]. But the problem of this study was that approach was demonstrated without implementation.

In 2019, [39] was further extended by Hall et al. [40] for the detection of no-sleep energy bugs using a reference count which keeps track of acquiring and releasing wakelocks. As a wakelock is acquired, the count is increased and counter is decreased when wakelock is released. However, one more case can exits where a wakelock is released before being acquired. In such case, count still results in a zero value, but no-sleep bug exists as the acquired wakelock is never released. Therefore, researchers used sequential reference count for keeping the track of not only acquiring and releasing wakelocks, but also their sequences. Inadequacy in

this study was that false positives were generated due to not considering program's control flow.

In 2019, Song et al. [41] proposed a static analysis technique for the identification of service usage inefficiency in Android applications. As, services cause the unnecessary resource occupation and/or energy consumption in Android applications. The proposed approach was implemented in an open-source tool "Serv-Droid" which is based on the static analysis based tool named "Soot". But the inadequacy in this study was that false positives were generated due to the static analysis.

2.2 Analysis and Comparison

The main focus of this study is to identify the energy bugs that are detected by existing static analysis based approaches, but those can be false positives (FPs). Thus, to effectively detect the energy bugs this study considers the static analysis approaches. Several energy issues are discussed in the existing studies related to resource leakage (i.e. missing deactivaton of Wi-Fi, GPS and Http) and wakelock bugs (missing deactivation of sensors/wakelocks).

Table 2.1 lists the comparison of static analysis techniques proposed in the literature where publication year, objective of the study, limitations and strengths are mentioned. Moreover, it is also mentioned in the evaluation column that existing studies evaluated their approach or not, if so then how many applications were collected for the evaluation.

Table 2.2 lists the analysis of existing static analysis based techniques with different aspects (i.e. types of energy bugs, false positives, false negatives and human involvement). Reason for selecting these parameters for analysis is also mentioned in the discussion below. Several energy problems related to resource leaks are discussed in literature.

Table 2.1: Comparison of Static Techniques

No.	Pub. Year	Objectives	Strengths	Weaknesses	Evaluation	Ref.
1.	2012	Detection of no-sleep energy bugs	• Possible no-sleep bugs in an application were automatically achieved. • No false negatives.	• There was no mechanism to automatically prune out false positives. • Expected invocation orders of event handlers need to specify by developer, which is impractical and lead to limited coverage of possible behaviors.	• No-sleep bug detector was evaluated on 86 apps. • FPs were generated in 13 out of 55 applications.	[32]
2.	2013	Detection of resource leaks in event handlers	• A tool Relda was developed and it can automatically analyze an application's resource operations and locate the resource leaks. • Resource leak summary and report was generated.	• Due to not considering control flow false negatives were caused. • Imprecise assumption of wakelock releasing points causes false positives.	• 55 real apps were selected and analysis precision was 88.0%, and recall was about 91.0%.	[33]

No.	Pub. Year	Objectives	Strengths	Weaknesses	Evaluation	Ref.
3.	2016	Detection of resource leak problems	• Target was byte-code instead of source code because it can be easily modifiable by attackers to change behaviour of app. • Multi-thread technique was used to reduce the analysis time.	• False positives were generated due to static analysis. • Elimination of FPs was not easier. It can be easier by implementing more accurate analysis techniques e.g. symbolic or dynamic execution.	• Approach was evaluated with 103 real-world Android apps and 67 real resource leaks were found, that are confirmed manually.	[34]
4.	2016	Identification of wakelock leakage	• 8 patterns of wakelock misuses were identified. • Instead of making assumptions Elite automatically reasons about need of using wakelocks at different program points via dataflow analysis.	• 8 patterns of WL misuses were identified but only two patterns were considered. • Call graphs were statically constructed by using soot, these graph were imprecise. Therefore, certain wakelock misuses were missed and FPs or FNs were caused.	• Evaluated Elite on 12 versions of real-world subjects. • 6 of 12 versions were containing wakelock leakage issues.	[35]

No.	Pub. Year	Objectives	Strengths	Weaknesses	Evaluation	Ref.
5.	2016	Identification of energy bugs due to missing deactivation of listener events	• Static detection algorithm was proposed focusing on energy-related listener leaks pattern. • Cost of the analysis was low.	• Analysis is not general to represent multiple threads running concurrently with main UI thread. • Static detection has its own limitations, so false positives were caused.	• 17 known and new bugs were detected in applications.	[36]
6.	2016	Identification of energy-inefficiencies in graphic applications	• A method was evaluated for enhancing energy efficiency of graphics-intensive apps.	• False positives were generated due to static analysis that must be manually eliminated by user. • Call-back methods were excluded by using SPARK analysis. Therefore, it was not fully automatic and need to improve precision of static analysis.	• Evaluation was done by using three open source Android apps named, Freegemas, Zxzx and Wari.	[37]

No.	Pub. Year	Objectives	Strengths	Weaknesses	Evaluation	Ref.
7.	2016	Detection of resource leaks	• Open but not used problem is taken into considerations. • Protocol was enriched with inappropriate behaviors that may cause energy leaks.	• Different paths to a node of CFG can obtain different states for the same resource but for simplicity path conditions were not considered so an infeasible path leads to false positive (FPs).	• Experiments were conducted on 100 real apps collected from F-Droid and Git-hub, and find that 18 applications have energy leaks.	[16]
8.	2017	Identification of resource leaks	• Static Application Analysis Detector analysed resource leak at background programs. • Analysed the effective paths that decrease number of paths to be analyzed made the analysis more efficient.	• This framework only analysed some common functions of call-back and event response functions. i.e. onCreate, onStart • False positives were generated due to static analysis that must be manually eliminated by user.	• 64 real practical Android apps were used for evaluating approach Static Application Analysis Detector (SAAD).	[38]

No.	Pub. Year	Objectives	Strengths	Weaknesses	Evaluation	Ref.
9.	2017	Detection of resource leaks	• Protocol was enriched with inappropriate behaviours that may cause energy leaks.	• Different paths to a node of CFG can obtain different states for the same resource but for simplicity path conditions were not considered so an infeasible path leads to FPs.	• Experiments were conducted on 100 apps from F-Droid, and test datasets from Relda and GreenDroid.	[21]
10.	2018	Detection of no-sleep energy bugs	• Reference counts were used for keeping track of acquiring and releasing wakelocks.	• The approach was not implemented just idea was proposed. • False positives were generated	• A study is demonstrated with case example.	[39]
11.	2019	Detection of no-sleep energy bugs	• The approach was implemented with case example by using sequential reference count to keep track of acquiring and releasing wakelocks and their sequences.	• False positives were generated due to not considering program's control flow.	• Refined banking application is used to demonstrate this study.	[40]

No.	Pub. Year	Objectives	Strengths	Weaknesses	Evaluation	Ref.
12.	2019	Detection of service leakage	• Static analysis technique, ServDroid was presented to effectively detect service usage inefficiency bugs. • In-depth study of Android services was conducted. • Services can keep running even when the device screen is shut down. Therefore, a technique was presented to automatically and effectively detect the service leakage.	• The approach was implemented in an open-source tool ServDroid based on Soot. • False positives were generated due to static analysis.	• Analysis was conducted on 1,000 apps downloaded from Google Play. • Results indicate that 825 (82.5%) apps involve at least one type of service usage inefficiency bugs.	[41]

Table 2.2: Analysis of Static Techniques with different aspects

No.	GPU	No-sleep bugs/ Wake-locks	Resource leak	Human Involve-ment	False Positives	False Negatives	Ref.
1.		✓		✓	✓		[32]
2.			✓		✓	✓	[33]
3.			✓	✓	✓		[34]
4.		✓			✓	✓	[35]
5.			✓		✓		[36]
6.	✓			✓	✓		[37]
7.			✓		✓		[38]
8.		✓	✓		✓		[16]
9.			✓		✓		[21]
10.		✓			✓		[39]
11.		✓			✓		[40]
12.			✓		✓		[41]

After the critical analysis of the detection of energy bugs in Android application, it is observed that mobile phones lead to short battery due to not closing the resource after its acquisition. Another reason is unclosed-ness where acquired resource is used but not closed. One more situation of energy bugs is observed that is unused-ness where the acquired resource is closed without any use. The current state of-the-art depict that most of the existing studies have focused on unclosed-ness situation but very few studies focused on unused-ness. In other words, due to energy bugs in Android applications the mobile battery depleted in a short time. Energy bugs arise due to Android applications developers as they could get difficulty while writing the code and may acquire the resources and forget to release which lead to severe battery drain.

Several approaches exist in literature for the detection of energy bugs in Android applications. After studying the existing techniques, we conclude that all existing studies are using different techniques for the detection of energy bugs in Android

applications. Some studies are based on static and dynamic analysis while other studies are based on hybrid analysis for energy bugs detection in Android applications. Existing dynamic and hybrid techniques generate the false positives and some of them generate the false negatives due to different reasons. Dynamic approaches are execution based. Due to the execution of code dynamic approaches are able to detect energy bugs accurately. However, due to the execution of test cases these techniques are more costly. Moreover, hybrid techniques are the combination of static and dynamic analysis and therefore provide the advantages of both static and dynamic analysis. In contrast, static approaches are non-execution based. These approaches consider the static analysis of source code for detecting the energy bugs in Android applications. Static analysis is able to examine all the program paths which make it valuable. However, it is not able to evaluate the conditions of source code and there may be paths that are infeasible or invalid. Therefore, due to not executing the applications source code static analysis based techniques generate more number of false positives. It can be said that false positives consequences are frequently obtained because sometimes those paths are also analysed by static analysis that are considered infeasible or in other words, paths that fail to execute under any input or test cases. Besides, it is concluded that from static approaches some of the paths are identified as bug prone that are not indicating the real energy bugs (i.e. infeasible paths). Infeasible paths lead to false positives. So, it is clear that infeasible paths lower the accuracy of the results, especially the ones that are assumed to have energy bugs. That's why, in static analysis approaches, false positives problem is a major issue that must be addressed. As Marashdih et al. [42] stated that for the enhancement of results of static analysis, there is a need to remove the infeasible paths from the entire paths of control flow graph (CFG). Any path within the CFG that fails to execute under any input value or test case is considered as infeasible path [43]. Researchers stated that presence of infeasible paths is a challenging problem, as there is no input for these paths to be executed [44].

Most of the static analysis based studies focus on different types of energy bugs i.e. resource leaks, wakelock bugs. Moreover, in existing state-of-the-art, due to

false positives the quality of results is reduced. Additionally, the main parameter that is used for evaluation in exiting static studies is false positives as due to the limitation of static analysis generation of false positives is obvious. Different existing static studies including [16],[21],[32],[33],[34],[35],[36],[37],[38] evaluate there techniques by using the parameter of false positives. Some studies identified the number of false positives through manual validation [16],[21]. Human involvement is impractical in some context. As Pathak et al. [32] stated that human involvement lead to limited coverage. On the other hand, some studies [33],[35] include the parameter of false negatives as from their approach false negatives are also generated due to different reasons. Therefore, different parameters for evaluating the existing studies i.e. false positives, false negatives and human involvement are considered. Further, we argue that there is a need to improve the quality of results by eliminating the number of false positives.

Chapter 3

Proposed Approach

The primary observations from literature review that motivated and signify our proposed approach are as follows:

1) Static approaches generate more number of false positives (FPs).

2) Dynamic approaches are more costly due to the execution of test cases as there can be large number of paths that need to be analysed.

3) Hybrid approaches can results in extra execution time due to dynamic analysis.

We proposed a static analysis based approach to address the issues of existing static analysis approaches i.e. infeasible paths lower the accuracy of the results. The proposed approach performs the identification of energy bugs and further evaluation of infeasible paths by using the constraint solver.

In this chapter, the proposed approach is described for better detection of energy bugs in Android applications. Flowchart of our proposed approach is shown in Figure 3.1. Moreover, main processes of proposed approach are explained in the following sections. Whereas, working of the proposed approach is demonstrated with an example application. The example application is based on the URL checking by using the "Wi-Fi" resource where Google homepage will be open if URL will be valid. Otherwise, it will show that URL is invalid.

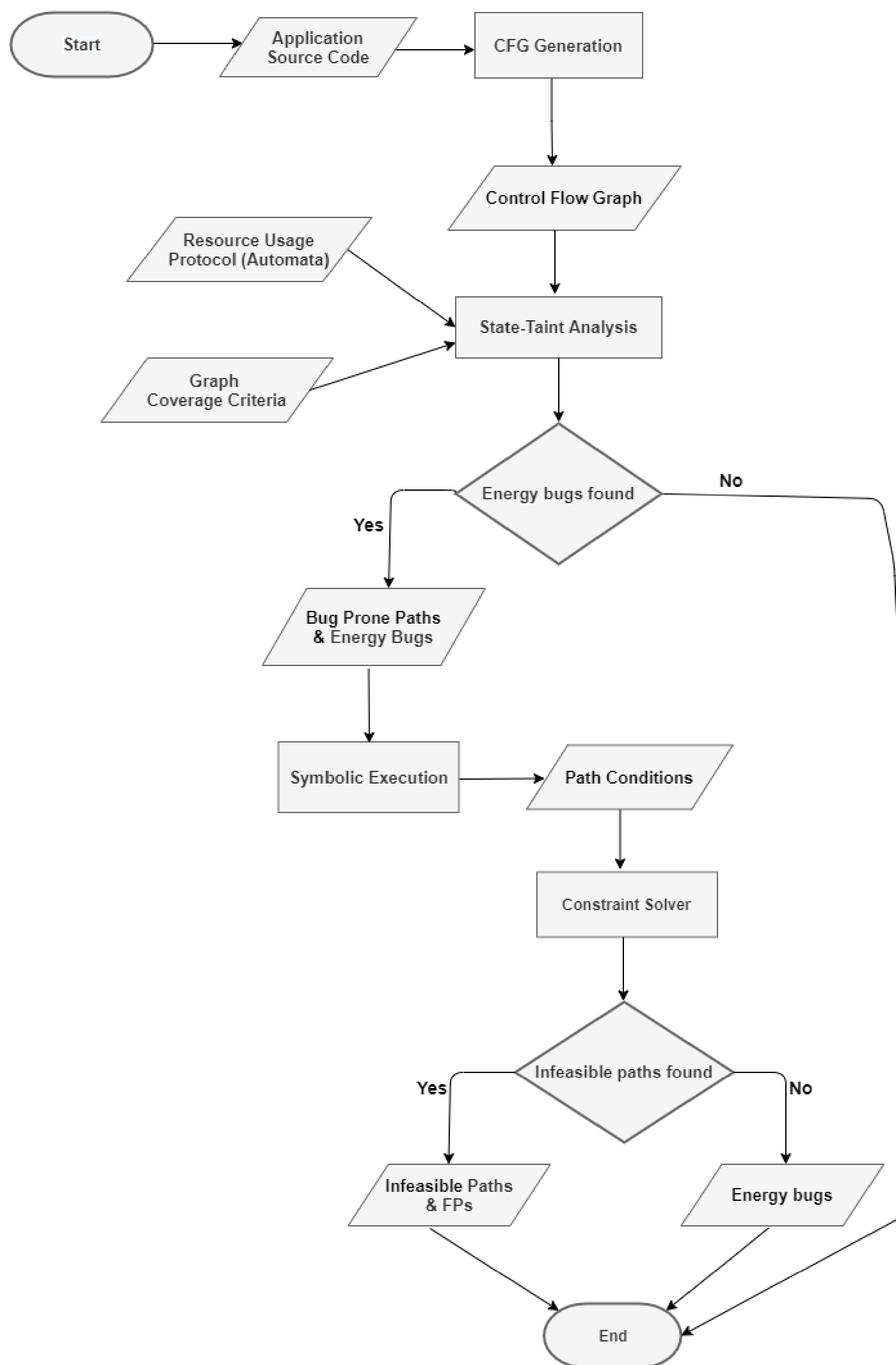

FIGURE 3.1: FlowChart of Proposed Approach

3.1 Control Flow Graph Generation

For comprehensive evaluation of our proposed approach, the generation of control flow graph is a very crucial step. As the researchers stated that control flow graph (CFG) is the standard method of representing the flow of program [45].

Model of the program's flow is represented by a graph known as Control Flow Graph (CFG) of a program [19]. CFG is the combination of nodes and edges. Every sentence of a program is represented using a node and to represent a program's data flow nodes are linked together by edges [42]. The process starts with the starting node, program flow starts and each node (statement of program) is represented until the program ends. For the identification of energy bugs we have read the applications source codes and the CFG's are generated from application source code.

The following example is used to define the concepts of our proposed energy bugs detection technique. Figure 3.2 elucidates the Android application source code and Figure 3.3 illustrates the CFG for that example.

```
53        wifiSwitch.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
54
55            @Override
56            public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
57                if(isChecked) {
58                    wifiManager.setWifiEnabled(true);
59                    String url1 = "http://www.google.com";
60                    if (isValid(url1)){
61                        Intent browserIntent = new Intent(Intent.ACTION_VIEW, Uri.parse("http://www.google.com"));
62                        startActivity(browserIntent);
63                        }
64                    else{Toast.makeText(getApplicationContext(),"Url is invalid",  Toast.LENGTH_LONG).show();  }
65                String url = "http://www.google.com";
66                if  (isChecked && url.startsWith("http://") || url.startsWith("https://")){
67                    wifiManager.setWifiEnabled(false);
68                    return; }
69                }
70                else {Toast.makeText(getApplicationContext(),"Wifi is currently OFF",  Toast.LENGTH_LONG).show();  }
71                }});}
```

Figure 3.2: An example code

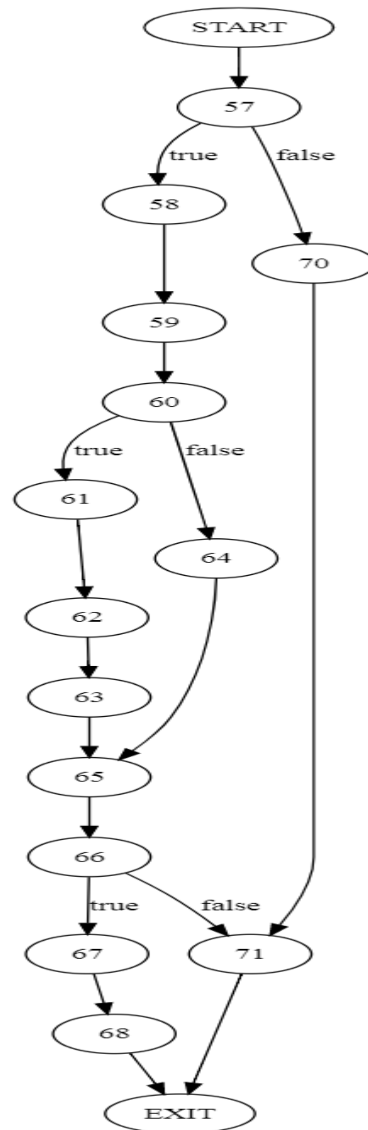

FIGURE 3.3: CFG of example code

Figure 3.2 and 3.3 shows that static analysis instigates by analysing the application source code and creating the CFG of the source code. Therefore, program's flow can be determined.

The Android application source code first check the condition specified in line 57, if condition is true it means Wi-Fi is enabled and the statement in line 58 will be implemented. It then ends at line 59 and next condition at line 60 will be checked that URL is valid or not. Likewise, if condition at line 60 is true then next statements (60, 62) will be implemented and Google homepage will be open. And if condition is false, line 64 will be implemented. At the end, condition at

line 66 will be checked, if it is true next 2 lines will be instigated and Wi-Fi will be turned off. Otherwise, program will be exited. However, if condition at line 57 is false, it will indicate that Wi-Fi is not enabled and line 70 will be implemented.

3.2 State Taint Analysis

For the detection of energy bugs in Android applications, we have applied the state-taint analysis. A static analysis named "State-taint analysis" was proposed by Xu el al. [16], in which control flow graph of a program was taken as input and guided by the resource protocols to track the resource actions among CFG. The idea of taint-analysis [46] was followed in current study. This analysis was capable to check the resource actions that either resource actions confirm to the corresponding resource protocols (i.e. automata) or not. A resource protocol specifies how a resource should be used or which action sequence is appropriate according to their usage protocol. Resource protocols can be represented as finite state automata (FSA). There are three possible states that are acceptable for the open but not used problem, named as i/c (i.e. the initial or closed/released state of resource), o (i.e. the open or acquired state of resource), and use (i.e. the resource is used).

For the general resource protocol an abstract automaton with energy leaks is given in the Figure 3.4. In Figure, O, U and C represent the actions designating the relevant APIs of resources opening or acquiring, using, and closing or releasing. An action which is according to the resource protocol will be considered appropriate. Conversely, any action which is not satisfying the resource protocol will lead to resource bug or energy bug i.e. resource is open but not closed, or resource is open, closed but not used. In other words, one use action (i.e. resource must be used before closing) between the open and close actions is necessary for being appropriate action. Moreover, there is no use action for the Wakelock. Wakelock have only open and close actions as shown in Figure 3.5. Furthermore, energy leak (el) indicates that the resultant action can cause an energy leak. Moreover, if a

resource is opened it should not be opened again, until the previous one is not used and if it is still remains open after use then it will consume energy. Consequently, a resource should be used after opening and before closing. Otherwise, opening unnecessary resource, or if a resource remains open after using could cause energy leaks.

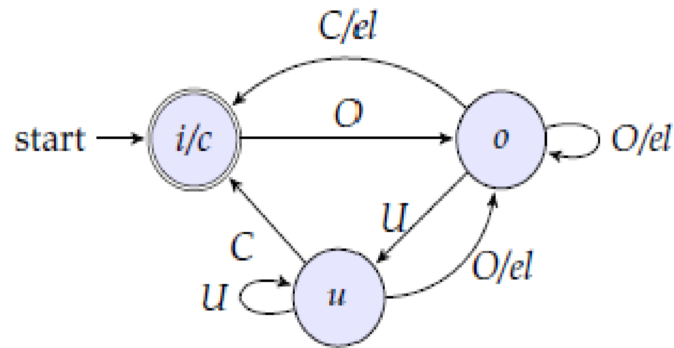

FIGURE 3.4: Abstract FSA for General Resources

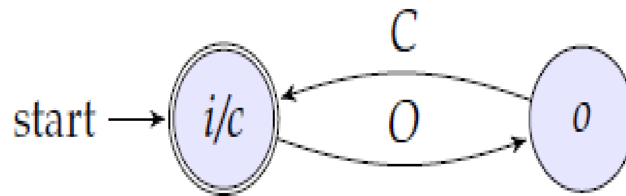

FIGURE 3.5: Specific FSA for Wakelock

Among many characteristics, there were also some limitations of this analysis i.e. the analysis was not able to completely detect all the paths of the control flow graph, as they were covering all the nodes of graph. Consequently, some code coverage was missed in the analysis due to incomplete coverage. As one resource can leads to different states from different paths, therefore, it is necessary to cover all the edges of the graph. Different graph coverage criteria's can be applied on control flow graph i.e. node coverage, edge coverage, edge pair coverage, loop coverage etc. We have applied graph coverage criterion named "Edge Coverage Criteria" to cover all the edges of the control flow graph.

Moreover, the enhanced analysis takes the control flow graph (CFG), and graph coverage criterion as input, and guided by the resource protocols for tracking the resource actions among control flow graph. According to the graph coverage criteria (Edge coverage criterion) control flow graph will be traversed and test paths will be generated. A path is a finite sequence of nodes connected with the edges of the control flow graph. Every path that starts from the starting node and goes until the exit node is known as test path in the program. The states of resources have been checked among the control flow graph, and our analysis checks for the resources actions whether they are according to the corresponding resource protocol or not (i.e. automata). In other words, our analysis verifies that whether all the resource actions follow the resource protocols or not.

Furthermore, some paths contain energy bugs such as if resources are acquired but not released, or released without being used on some paths then those paths will be consider as buggy or erroneous. Therefore, to observe the paths different graph coverage criteria's can be applied on the control flow graph. The test requirement for any coverage criteria is that it should cover all the requirements. We have applied the edge coverage criterion which means it should cover all the edges of the graph.

To define the concepts of enhanced state-taint analysis the "Edge coverage criterion" is applied on the CFG (shown in Figure 3.3). The Table 3.1 listed below shows the difference between edge and node coverage criterion. Moreover, states of resources have been checked, such as, the resource Wi-Fi example code shown in the Figure 3.2. CFG of example code is generated and according to CFG, Wi-Fi is acquired at edges (57, 58), used at (60, 61) and closed at (66, 67).

Furthermore, according to the edge coverage criterion, formed test paths, identified buggy paths and details of Wi-Fi resource (by taking automata shown in Figure 3.4 as a guide) are shown in the Table 3.2. Moreover, T and F show the true and false values of the conditions, if any conditions in source code will be true and false it will be considered as T and F respectively.

Table 3.1: Graph Coverage Criteria's

Graph Coverage Criteria's	Node Coverage	Edge Coverage
Test Requirements	{57,58,59,60,61,62,63,64, 65,66,67,68,70,71}	{(57,58), (57,70), (70,71), (58,59), (59,60), (60,61), (60,64), (61,62), (62,63), (63,64), (63 ,65), (65 ,66), (66,67), (66,71), (67,68) }
Test Paths	[57,58,59,60,61,62,63,65,66, 67,68] //TTT [57,58,59,60,64,65,66,67, 68] //TFT [57,70,71] //F	[57,58,59,60,64,65,66,71] //TFF [57,58,59,60,61,62,63,65,66,67,68] //TTT [57,70,71] //F [57,58,59,60,61,62,63,65,66,71] //TTF [57,58,59,60,64,65,66,67,68] //TFT

Table 3.2: Test paths, Buggy paths and Resource details

No.	Test Paths	Resource (Wi-Fi) details	Buggy paths
1.	[57,58,59,60,64,65,66,71]	Wi-Fi is opened, not used not closed. (TFF)	✓
2.	[57,58,59,60,61,62,63,65, 66,67,68]	Wi-Fi is opened, used and closed. (TTT)	
3.	[57,70,71]	Wi-Fi is not opened. (F)	
4.	[57,58,59,60,61,62,63,65, 65,66,71]	Wi-Fi is opened, used but not closed. (TTF)	✓
5.	[57,58,59,60,64,65,66,67,68]	Wi-Fi is opened, not used but closed. (TFT)	✓

3.3 Symbolic Execution

Static approaches perform static analysis of code to detect the energy bugs without executing the source code of applications. Static analysis is appreciated because of the aptitude to examine all possible paths of program. As the Marashdih et al. [18] stated that to analyse the source codes, static analysis has been commonly used.

Conversely, there are more possibilities of false positives because code is not executed and infeasible paths lead to results in false positives. In the static analysis, false positives consequences are frequently obtained because sometimes those paths are also analysed by static analyses that are considered infeasible. Infeasible paths can exist due to several reasons, one of which is the existence of dead code, other one is conflicting clauses that are contained within certain paths. That's why, in static analyses, false positives results problem is a major issue that must need to be addressed. So, to overcome the problem of false positives generation, we need to check the identified bugs that either they were false positives or actual bugs. For this purpose we have used the symbolic execution. Symbolic execution is a well-known program analysis technique for the identification of infeasible paths within the source code.

As researchers stated that symbolic execution was first introduced in the 1970s and it is one of the gradually popular program analysis technique [47]. Cadar et al. [48] stated that in the recent years, symbolic execution has gain interest due to the constraint solving technology and availability of computational power. Moreover, it is a program analysis technique which executes the program with the symbolic inputs rather than concrete inputs and represents the values of program variables as symbolic expressions.

Furthermore, it maintains the path condition that is updated whenever branch instruction is executed, to encode the constraints on the inputs that reach that program point. Moreover, for every analyzed program path it checks that whether all possible executions of the path are not violating the specification of the program by generating a condition (logical formula) for all possible executions. If the formula is falsifiable, specification can be violated. Besides, Symbolic execution can be tremendously expensive due to lots of possible program paths and to decide which path is feasible or infeasible need to query solver a lot [48]. There can be complexity issues due to constraints that are generated. On the other hand, there are scalability issues due to enormous number of paths that need to be analyzed. Figure 3.6 shows the path conditions for the identification of false positives in Android application source code (shown in Figure 3.2).

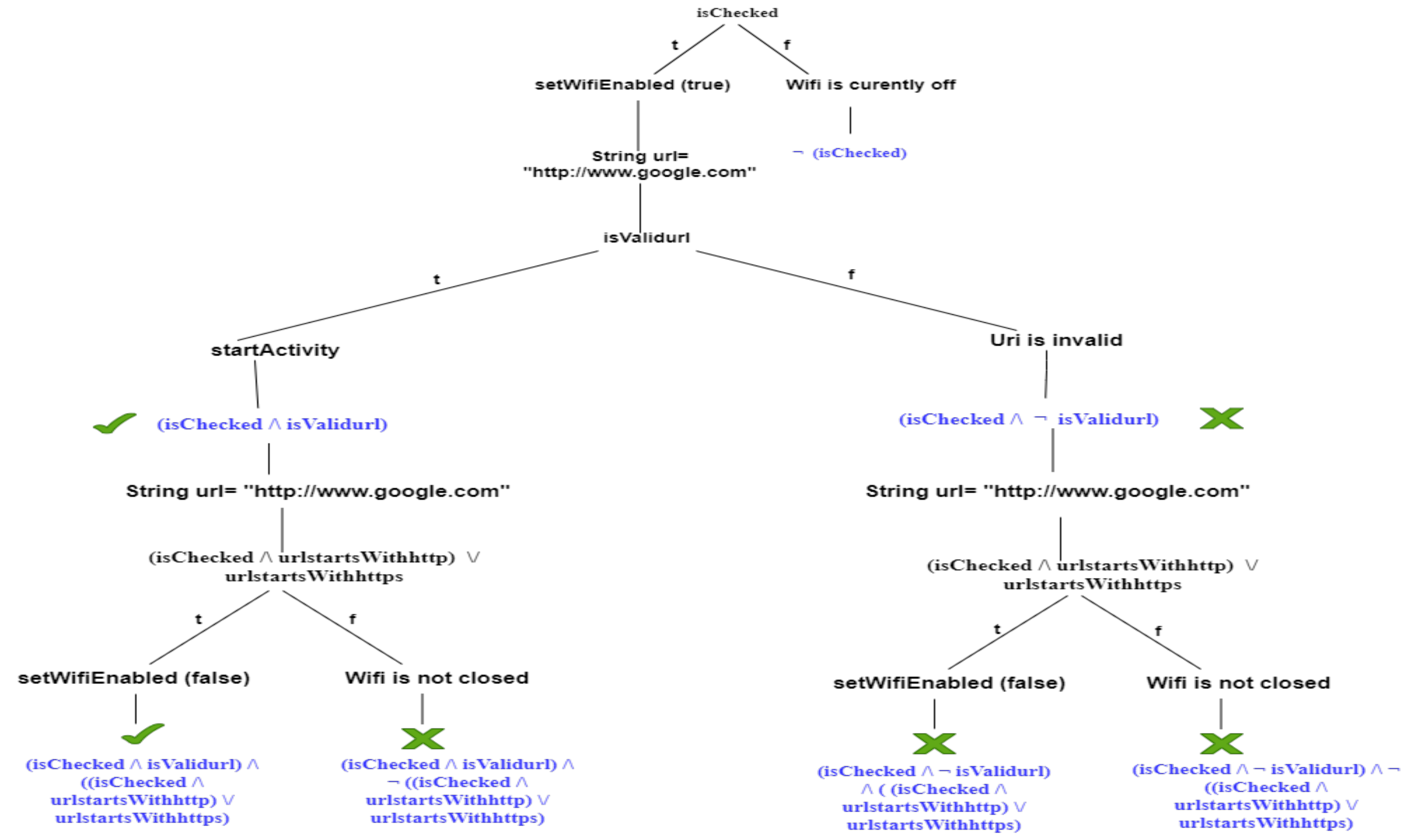

FIGURE 3.6: Path Conditions for example code

3.4 Constraint Solver

For program analysis, symbolic execution has recently become practical due to advances in constraint solvers [49]. Due to static analysis, FPs can be obtained because sometimes the paths that are not executable (also called infeasible paths) can be considered. Infeasible paths lead to false positives, which is a major issue in static analyses that needs to be addressed. The paths that identify the existence of energy bugs from static analysis can be FPs. To identify FPs we have added path conditions for each erroneous path to check whether these paths are identifying real energy bugs or FPs.

Furthermore, one program path is analysed at a time and then possible bugs in that path are checked. Symbolic execution checks for each analysed program path whether all possible execution of the program are according to the specifications of program or not. This can be done by generating constraint or condition for all possible executions. If the constraint is falsifiable it means specifications are violated. Constraint solver is also known as query solver for the evaluation of the path conditions and generating the accurate results. By using the constraint solver those paths can be identified which contains real energy bugs as well as FPs can also be identified which are generated from static analysis approach. Moreover, after applying the path conditions and generating a condition (logical formula or constraint), infeasible paths can be identified. If there is at least one input for the execution of program path, then that path will be feasible. In other words, the path will be feasible if the logical formula is satisfiable that represent the program path, otherwise it will be infeasible [50]. Infeasible paths are not able to execute under any input [44].

Besides, as shown in Table 3.2 three paths are identified as bug prone. Further, we have used the symbolic execution to validate that these paths are indicating real energy bugs or these are false positives. Moreover, it is identified that static analyses were not able to accurately detect energy bugs because code was not executed. Therefore, if there is any path indicating the existing of bug from static analysis; but that path are not executed at all then bug will not actually arise

and will be considered as FPs. In other words, when program is executed, there is possibility that some conditions will never be executed or if executed they may indicate those paths were not bug prone. Such that, if condition at (Line 57) is true then condition at (Line 60) will never be false because if Wifi is enabled on device and correct URL is mentioned at (Line 59) then URL will never be incorrect. So, path 1 and 5 that were indicating bug from static analysis will be considered as FPs. Also, if both conditions at node 57 (57, 58) and at node 60 (60, 61) are executed then condition at node 66 (66, 67) must be executed because there is OR operator that will show that if one condition is satisfied, Wifi will be closed. So, path 4 that was indicating bug from static analysis will also be considered as FPs. Therefore, we focused on static analysis for detecting the infeasible paths that leads to FPs. Symbolic execution is used for each buggy path, and path conditions are created to identify the FPs.

Constraint solver evaluates the path conditions which are shown in Table 3.3 and results show that all the buggy paths revealed in Table 3.2 are FPs. Whereas, T and F are the values of conditions where T shows the true values and F shows the false values.

Table 3.3: Path conditions and Results of buggy paths

No.	**Buggy paths**	**Path Conditions**	**Results**
1.	TFF	(isChecked /\ ¬isValidurl) /\ ¬ ((isChecked /\ urlstartsWithhttp) \/ urlstartsWithhttps)	False positive
2.	TTF	(isChecked /\ isValidurl) /\ ¬ ((isChecked /\ urlstartsWithhttp) \/ urlstartsWithhttps)	False positive
3.	TFT	(isChecked /\ ¬isValidurl) /\ ((isChecked /\ urlstartsWithhttp) \/ urlstartsWithhttps)	False positive

Chapter 4

Implementation

In the previous chapter, we have explained the details of the proposed methodology. This chapter presents the details about the implementation of our proposed approach. We have implemented our approach in java language using eclipse Mars.2 software. Furthermore, control flow graphs are generated using Eclipse plugin "CFG factory. Likewise, symbolic execution is used and for the evaluation of path conditions a constraint solver named "MiniZinc" is used.

In this chapter, different steps of implementation are discussed as follows.

4.1 Control Flow Graph Generation

Control flow graph is the graph, whose nodes represent a statement of the program and the edges represent the flow in which these statements are executed. As many innovative Android applications are developed by developers due to the Android's popularity among users. Moreover, Google Play official Android app market hosts more than three million apps with large number of downloads each day. Therefore, we have chosen Android.

First of all, we have downloaded the source codes of different real world Android applications from well-known repositories i.e. Git-hub and Google code. After

that, we have read the applications source codes and generated the control flow graph of each application. We have used eclipse plugin named as "CFG factory plugin" for the control flow graph generation. A program which generates the control flow graphs from java byte code is known as CFG factory [51]. Generated graphs can be further modified as well as exported to DOT, XML and several raster image formats.

Firstly, we have read the application source code and then explored the energy bugs (i.e. resources that are acquired, used but not closed, or acquired, closed but never used). After exploration of energy bugs we have created control flow graph for that particular function in which we have found the energy bugs. Besides, control flow graphs are exported to XML format and Graph coverage criterion (Edge coverage) is applied on control flow graph. Moreover, for visualization the graph we have used GraphViz after exporting the graph in DOT format.

As an example, Figure 4.1 elucidates the Android application source code and Figure 4.2 illustrates the control flow graph for that example.

XML and DOT of the control flow graph (shown in Figure 4.2) are shown in Appendix.

```
75  private void findMyLocation(Context context)
76  {LocationManager manager=(LocationManager) context.getSystemService(Context.LOCATION_SERVICE);
77  Criteria criteria=new Criteria(); criteria.setAccuracy(Criteria.ACCURACY_FINE);
78  String provider=manager.getBestProvider(criteria, true);
79  Location location = manager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
80  boolean isGPSEnabled = manager.isProviderEnabled(LocationManager.GPS_PROVIDER);
81      if(isGPSEnabled)
82      { location=manager.getLastKnownLocation(provider);
83      if(location!=null)
84      { latitude=location.getLatitude();
85      longitude=location.getLongitude();}
86      if(location!=null || isGPSEnabled){
87          manager.removeUpdates((LocationListener) this);
88          Toast.makeText(context, "GPS closed for this app!", Toast.LENGTH_SHORT).show();
89          return;} }
90      else {Toast.makeText(context, "gps is not enabled!", Toast.LENGTH_SHORT).show();}
91      }}
```

Figure 4.1: An example code

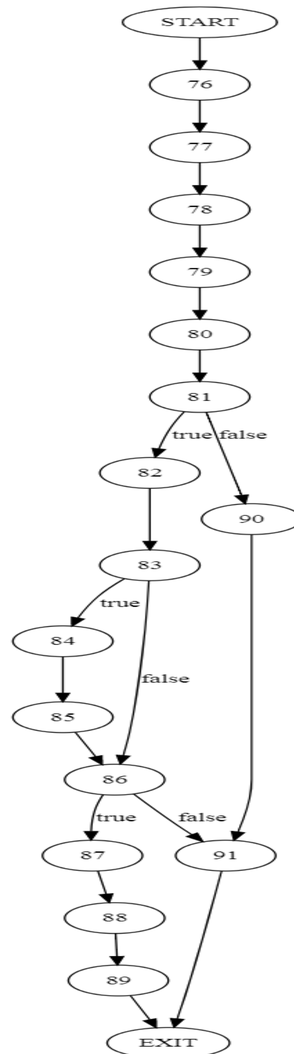

FIGURE 4.2: CFG of example code

Figure 4.1 and 4.2 shows that static analysis instigates by analysing the application source code and creating the CFG of the source code. Therefore, program's flow can be determined.

The Android application source code first check the condition specified in line 81, if condition is true it means GPS is enabled and the statement in line 82 will be implemented. Then, next condition at line 83 will be checked that location is null or not. At the end, condition at line 86 will be checked, if it is true next 3 lines will be instigated implemented and GPS will be closed for particular application. Otherwise, program will be exited. However, if condition at line 81 is false, it will indicate that GPS is not enabled and line 90 will be implemented.

4.2 State Taint Analysis

State taint analysis is applied for the purpose of energy bugs detection in Android applications. CFG of application source code and edge coverage criterion have been taken as input by taking general automata of resources (shown in Figure 3.4) as a guide to create the bug prone test paths as output. CFG is generated by using the eclipse Plugin "CFG factory". After that CFG is exported to XML (shown in Appendix). Moreover, graph coverage criterion is applied by taking the XML format of CFG and test paths are generated. Afterwards, general automata of resources (shown in Figure 3.4) is taken into consideration and an action that does not satisfy its corresponding protocol (i.e. if action sequence is not according to resource automata) is considered as a resource usage bug i.e. resource is open but not closed or in other situation where resource is open, closed but not used.

Control flow graph of example code and resource automaton are shown in Figure 4.2 and Figure 4.3 respectively. The "Edge Coverage Criterion" is applied on the control flow graph (shown in Figure 4.2). Table 4.1 listed below shows the difference between edge and node coverage criterion.

TABLE 4.1: Graph Coverage Criteria's

Graph Coverage Criteria's	**Node Coverage**	**Edge Coverage**
Test Requirements	{76,77,78,79,80,81,82,83, 84,85,86,87,88,89,90,91}	{(76,77), (77,78), (78,79), (79,80), (80,81), (81,82), (81,90), (82,83), (83,84), (83,86), (84,85), (85,86), (86,87), (86,91), (87,88), (88,89) }
Test Paths	[76,77,78,79,80,81,82,83, 84,85,86,87,88,89] //TTT [76,77,78,79,80,81,90,91] //F [76,77,78,79,80,81,82,83, 84,85,86,91] //TTF	[76,77,78,79,80,81,82,83,84,85,86, 87,88,89] //TTT [76,77,78,79,80,81,82,83,84,85,86,91] //TTF [76,77,78,79,80,81,82,83,86,87,88,89] //TFT [76,77,78,79,80,81,82,83,86,91] //TFF [76,77,78,79,80,81,90,91] //F

Moreover, the test requirement for any coverage criteria is that it should cover all the requirements. By applying node coverage, it will cover all the nodes of the graph but it does not provide complete coverage because it does not cover all edges of the graph. Therefore, we have applied the edge coverage and it is able to cover all the edges of the graph. As Table 4.1 shows that, from applying node coverage on the control flow graph, there is possibility that some edges may be missed such as (83, 86). Furthermore, edge coverage is able to completely detect all the edges of the control flow graph, which represents the program's flow. In addition, states of resources have been checked, such as, the resource GPS example code shown in Figure 4.1. Control flow graph of example code is generated and according to control flow graph, GPS is acquired at edges (81, 82), used at (83, 84) and closed at (86, 87). Furthermore, according to the edge coverage criterion, formed test paths, identified buggy paths and details of GPS resource (by taking automata shown in Figure 3.2 as a guide) are shown in Table 4.2.

TABLE 4.2: Test paths, Buggy paths and Resource details

No.	**Test Paths**	**Resource (GPS) details**	**Buggy paths**
1.	[76,77,78,79,80,81,82,83, 84,85,86,87,88,89]	GPS is opened, used and closed. (TTT)	
2.	[76,77,78,79,80,81,82,83, 84,85,86,91]	GPS is opened, used but not closed. (TTF)	✓
3.	[76,77,78,79,80,81,82,83, 86,87,88,89]	GPS is opened, not used but closed. (TFT)	✓
4.	[76,77,78,79,80,81,82,83, 86,91]	GPS is opened, not used, not closed. (TFF)	✓
5.	[76,77,78,79,80,81,90,91]	GPS is not opened. (F)	

By taking the resource usage protocol (resource automata) as guide, test paths are identified as buggy or bug free. As we can see that three paths are identified as bug prone. Further, we have used the symbolic execution to validate that these paths are indicating real energy bugs or these are false positives.

4.3 Symbolic Execution

Symbolic execution is gradually popular program analysis technique [47]. The program is explored systematically by analysing one program path at a time and then possible bugs in that path are checked. Besides, after applying the state taint analysis, different paths are identified. Some of them are having energy bugs and some of them are appropriate paths (i.e. bug free paths) or paths that are not indicating energy bugs.

According to researchers [48], due to the enormous number of paths that need to be analyzed there are scalability issues faced by symbolic execution. Additionally, another issue is the complexity of the generated constraints. Therefore, to overcome this issue, the proposed approach performs symbolic execution only on erroneous paths (i.e. paths that are indicating energy bugs from state taint analysis) not on all the identified test paths. The paths which are having energy bugs, for example bug prone paths, are chosen. Moreover, path conditions are applied on each bug prone path to find that identified paths are real energy bugs or false positives.

As shown in Table 4.3, we have considered only the buggy paths revealed in Table.4.2, and added the path conditions for the identification of infeasible paths that lead to false positives.

Table 4.3: Path Conditions for buggy paths

No.	**Buggy paths**	**Path Conditions**
1.	TTF	(isGPSEnabled /\ (Location!= null)) /\ (¬ (Location !=null \/ isGPSEnabled))
2.	TFT	(isGPSEnabled /\ ¬ (Location != null)) /\ (Location !=null \/ isGPSEnabled)
3.	TFF	(isGPSEnabled /\ ¬ (Location != null)) /\ (¬ (Location !=null \/ isGPSEnabled))

4.4 Constraint Solver

The static analyses have major issue of infeasible paths which leads to false positives. To identify false positives we have added the path conditions for each bug prone path to check whether these paths are identifying actual energy bugs or false positives. Furthermore, symbolic execution checks for each analysed program path, whether all possible executions of the path are not violating the specifications of the program by generating a condition (logical formula or constraint) for all possible executions.

To reduce the complexity of constraint, we have chosen only those paths that are indicated as energy bugs after applying the enhanced state taint analysis. Further, by using symbolic execution path conditions are created only for identified buggy paths. After that, these conditions are evaluated by using the constraint solver named "MiniZinc". It is also known as query solver to identify the infeasible paths. Moreover, it is used for solving Boolean expressions.

Following figures (Figure 4.3, Figure 4.4 and Figure 4.5) show the Constraint solver (MiniZinc) code and output of the MiniZinc for the path conditions that are created for buggy paths (shown above in Table 4.3).

```
1 var bool: isGPSEnabled;
2 var float: Location;
3 int: null=0;
4
5 isGPSEnabled = true;
6 Location = true;
7
8 constraint (isGPSEnabled /\ (Location!= null)) /\ (¬(Location !=null \/  isGPSEnabled));
9 solve satisfy;
10
```

```
Output

Running TTF.mzn
=====UNSATISFIABLE=====
Finished in 216msec
```

FIGURE 4.3: MiniZinc code and Output of buggy path (TTF)

```
1 var bool: isGPSEnabled;
2 var float: Location;
3 int: null=0;
4
5 isGPSEnabled = true;
6 Location = false;
7
8 constraint (isGPSEnabled /\ ¬(Location != null)) /\ (Location !=null \/ isGPSEnabled);
9 solve satisfy;
10
```

Output

```
Compiling TFT.mzn
Running TFT.mzn
isGPSEnabled = true;
Location = 0.0;
----------
Finished in 213msec
```

FIGURE 4.4: MiniZinc code and Output of buggy path (TFT)

```
1 var bool: isGPSEnabled;
2 var float: Location;
3 int: null=0;
4
5 isGPSEnabled = true;
6 Location = false;
7
8 constraint (isGPSEnabled /\ ¬(Location != null)) /\  (¬(Location !=null \/  isGPSEnabled));
9 solve satisfy;
10
```

Output

```
Compiling TFF.mzn

  WARNING: model inconsistency detected
Running TFF.mzn
=====UNSATISFIABLE=====
Finished in 214msec
```

FIGURE 4.5: MiniZinc code and Output of buggy path (TFF)

As shown an example code in Figure 4.1, the statement presented in line 86 is an infeasible statement because if above conditions at node 81 (81, 82) and at node 83 (83, 84) are executed then there will be no possibility for the given condition in line 86 to be false at all. There is OR operator that will show that if one condition is satisfied, GPS will be closed for this particular application. Thus, every path that has the line 86 false implementation will be considered an infeasible path. So, it is identified that 3 paths were detected as buggy paths and 2 of them are infeasible paths as detected by our approach.

Chapter 5

Results and Discussion

In the previous chapters, we have explained the in-depth details of the proposed methodology and its implementation. This chapter presents the details about the results that have been obtained by applying the proposed methodology and also these results are discussed in detail.

We have collected the real world Android applications and explored the energy bugs in each application. Moreover, we have generated a CFG by using Eclipse plugin named "CFG factory", and edge coverage criterion was applied on CFG's as discussed in previous chapter.

To test and address about the precision of our technique, we have conducted a series of experiments on 20 real Android applications to detect energy leaks, where the applications are collected from well-known open source Android application repositories i.e. Git-hub and Google code. Moreover, a test dataset of Statedroid [21] and some other applications are also considered for evaluation. All the experiments have been conducted on Eclipse mars.2 and MiniZinc, core I3 CPU and 4GB RAM, running Windows 8.1.

Besides, we have performed state-taint analysis on the source codes of those applications where energy bugs are reported. Further, energy leaks in each application were detected, by taking resource protocols as a guide for HTTP connection (Fig.3.4), WakeLock (Fig.3.5) and Media Player (Fig.3.4) respectively. Likewise,

bug prone paths are identified from state-taint analysis and path conditions are added for those paths. Our analysis verifies that whether all the resource actions follow the resource protocols or not and identify those bugs that were FPs.

5.1 Android Applications

To test our approach, we have applied the proposed approach on different Android applications source codes. We have identified the buggy paths after performing the state-taint analysis and path conditions are created for buggy paths. Thorough results of some of the Android applications are presented below.

First of all, an application named ”Adbwireless” source code (Figure 5.1), CFG (Figure 5.2), test paths (Table 5.1) and path conditions (Table 5.2) are presented below. Moreover, the MiniZinc code and output for the path conditions of buggy paths are presented in Figure 5.3 and Figure 5.4 respectively. The resource kind is Wakelock (WL). Wakelock is acquired at line 19, after that further conditions will be checked. If condition at line 30 is true then next statement at line 31 will be executed and WL will be released. If it is false, WL will not be released and it will indicate the existence of energy bug.

```
16 public static void adbStart(Context paramContext )
17       { PowerManager.WakeLock mwakeLock;
18       Context context = null; PowerManager pm = (PowerManager)context.getSystemService(Context.POWER_SERVICE);
19                mwakeLock.acquire(); }
20          public static boolean adbStop(Context paramContext)
21                   throws Exception
22                 {try{
23                     if (adbWireless.mState)
24                     {setProp("service.adb.tcp.port", "-1"); runRootCommand("stop adbd");
25                     runRootCommand("start adbd");}
26                     adbWireless.mState = false;
27                     ((SharedPreferences.Editor)localObject).putBoolean("mState", false);
28                     if (prefsAutoCon(paramContext))
29                       autoConnect(paramContext, "d");
30                     if (mWakeLock != null)
31                       mWakeLock.release();
32                     if (mNotificationManager != null)
33                       mNotificationManager.cancelAll();
34                     return true;}
35                   catch (Exception paramContext1){ }
36                 return false; }
```

FIGURE 5.1: Source code of Adbwireless Application

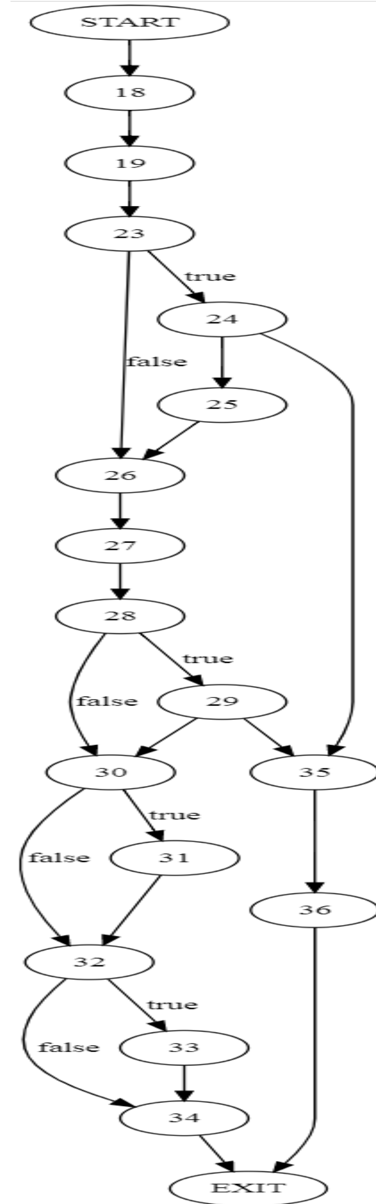

FIGURE 5.2: CFG of Adbwireless Application

TABLE 5.1: Test paths, Buggy paths and Resource details

No.	Test Paths	Resource (Wakelock) details	Buggy paths
1.	[Start,18,19,23,26,27,28,29, 30,32,33,34, Exit]	Wakelock is open but not closed. (FTFT)	✓
2.	[Start,18,19,23,24,25,26,27, 28,30,31,32,34,Exit]	Wakelock is opened and closed. (TFTF)	
3.	[Start,18,19,23,26,27,28,30, 31,32,34,Exit]	Wakelock is opened and closed. (FFTF)	
4.	[Start,18,19,23,26,27,28,30, 32,34,Exit]	Wakelock is opened but not closed. (FFFF)	✓

TABLE 5.2: Path Conditions for buggy paths

No.	Buggy paths	Path Conditions
1.	FTFT	¬(adbWirelessstate) /\ (prefsAutoCon) /\ ¬(mWakeLock != null) /\ (mNotificationManager != null)
2.	FFFF	¬(adbWirelessstate) /\ ¬(prefsAutoCon) /\ ¬(mWakeLock != null) /\ ¬ (mNotificationManager != null)

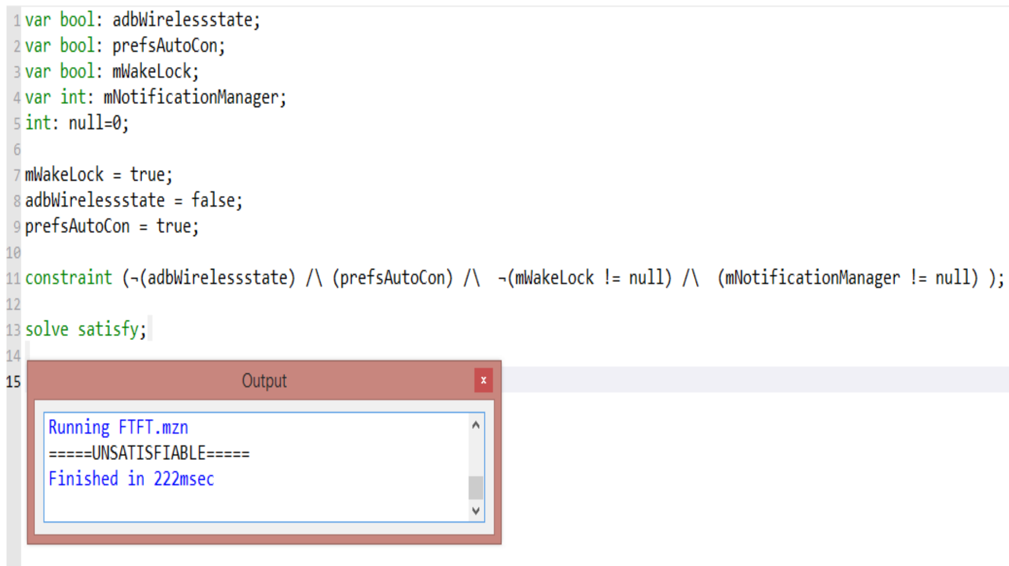

FIGURE 5.3: MiniZinc code and Output of buggy path (FTFT)

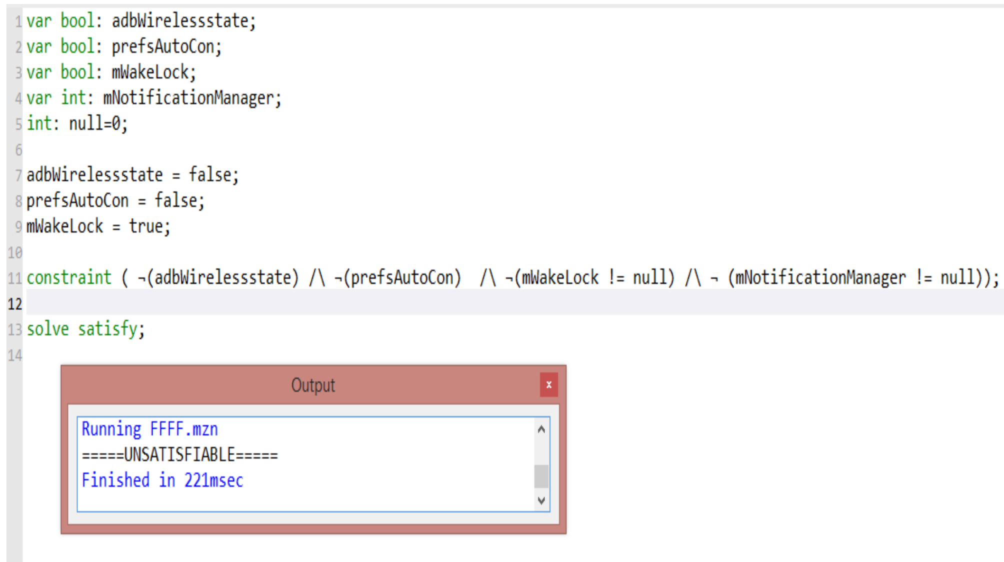

FIGURE 5.4: MiniZinc code and Output of buggy path (FFFF)

As shown in the Figure 5.1, the statement presented in line 30 is an infeasible statement as if once wakelock is acquired at line 19 then there will be no possibility for the given condition at line 30 to be false at all. Thus, every path that has the false implementation of line 35 will be considered an infeasible path. So, it is identified that two paths were detected as buggy paths and all of them are infeasible paths as detected by our approach.

Another application named ”Smspopup” source code (Figure 5.5), CFG (Figure 5.6), test paths (Table 5.3) and path conditions (Table 5.4) are presented below. Moreover, the MiniZinc code and output for the path conditions of buggy paths are presented in Figure 5.7, Figure 5.8 and Figure 5.9 respectively. The resource kind is media player (MP) that is acquired at line 29, used at line 33 and released at line 36. If the conditions at line 28 will be true then next statement at line 29 will be executed and media player will be acquired otherwise it will skip the next statement and check for the next condition at line 32. If it is true next line will be started and media player will be started for particular application. At the end, condition at line 35 will be checked, if it is true media player will be released eventually.

```
22 private static void buildNotification(Context context,String contactId,String contactLookupKey,
23         boolean onlyUpdate,int notif)
24         {
25         ManagePreferences mPrefs = new ManagePreferences();
26         try {
27             // Use MediaPlayer to play so they can hear the notification over the ear piece
28             if (mPlayer == null) {
29                 mPlayer = MediaPlayer.create(context, notifSoundUri);
30             }
31             // Check null again in case media-player couldn't be created
32             if (mPlayer != null) {
33                 ((MediaPlayer) mPlayer).start();}        }
34         catch (IllegalStateException e) {
35         finally {if (mPlayer != null) {
36                 ((MediaPlayer) mPlayer).release();}
37          }}
```

Figure 5.5: Source code of Smspopup Application

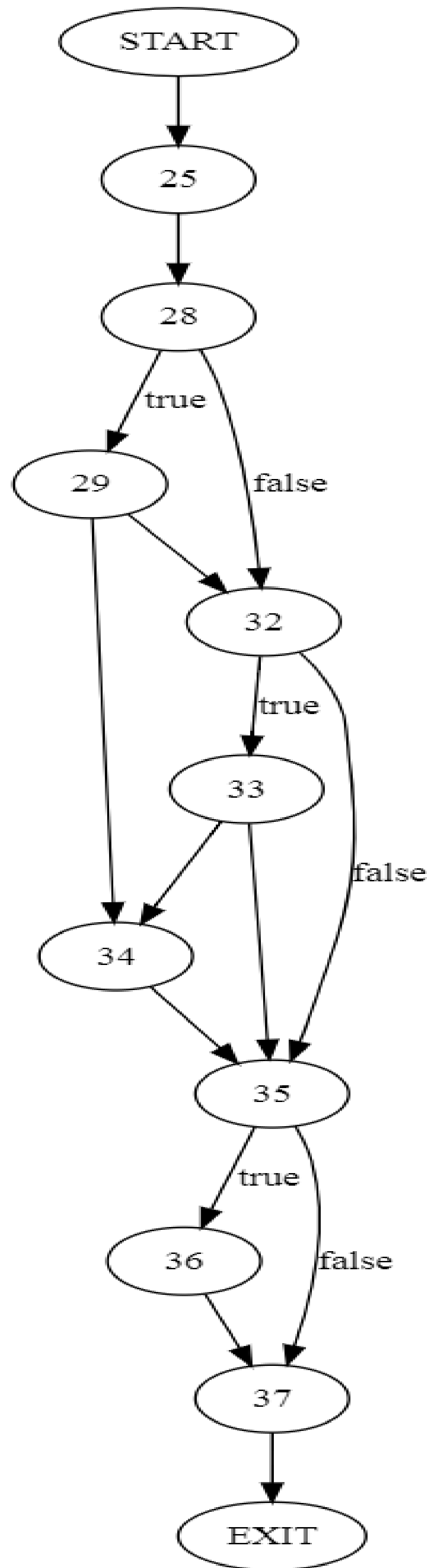

FIGURE 5.6: CFG of Smspopup Application

TABLE 5.3: Test paths, Buggy paths and Resource details

No.	Test Paths	Resource (Media player) details	Buggy paths
1.	[Start,25,28,29,32,33, 35,36,37, Exit]	Media player is open, used and closed. (TTT)	
2.	[Start,25,28,29,32,33, 35,37, Exit]	Media player is open, used but not closed. (TTF)	✓
3.	[Star,25,28,29,32,35, 36,37, Exit]	Media player is open, not used and closed. (TFT)	✓
4.	[Start,25,28,29,32,35, 37, Exit]	Media player is open, not used and not closed. (TFF)	✓
5.	[Start,25,28,32,35,37, Exit]	Media player is not open, not used, not closed. (FFF)	

TABLE 5.4: Path Conditions for buggy paths

No.	Buggy paths	Path Conditions
1.	TTF	(mPlayer == null) /\ (mPlayer != null) /\ not (mPlayer != null)
2.	TFT	(mPlayer == null) /\ not (mPlayer != null) /\ (mPlayer != null)
3.	TFF	(mPlayer == null) /\ not (mPlayer != null) /\ not (mPlayer != null)

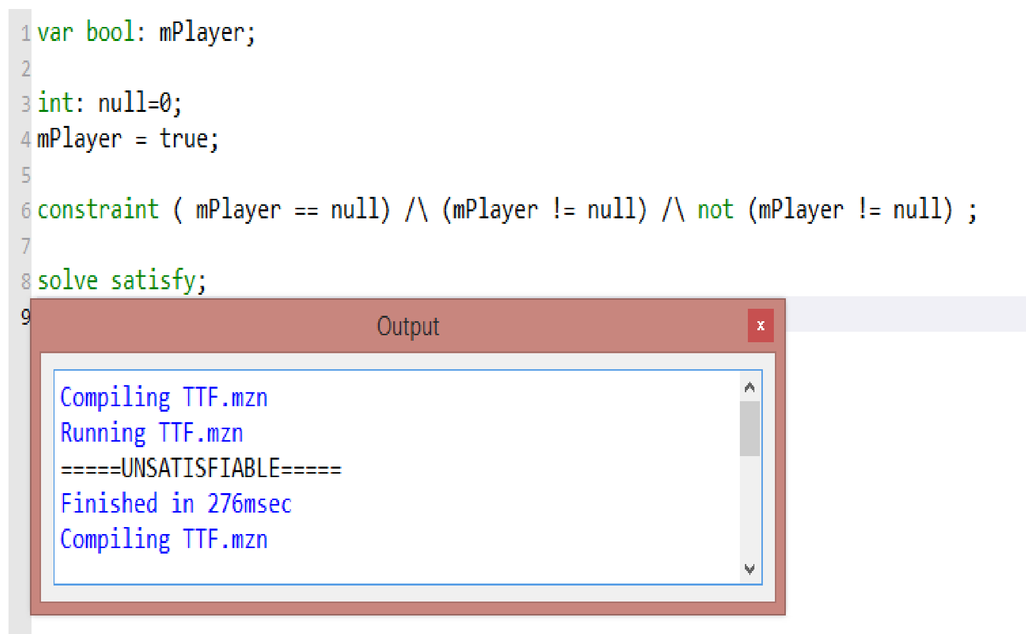

FIGURE 5.7: MiniZinc code and Output of buggy path (TTF)

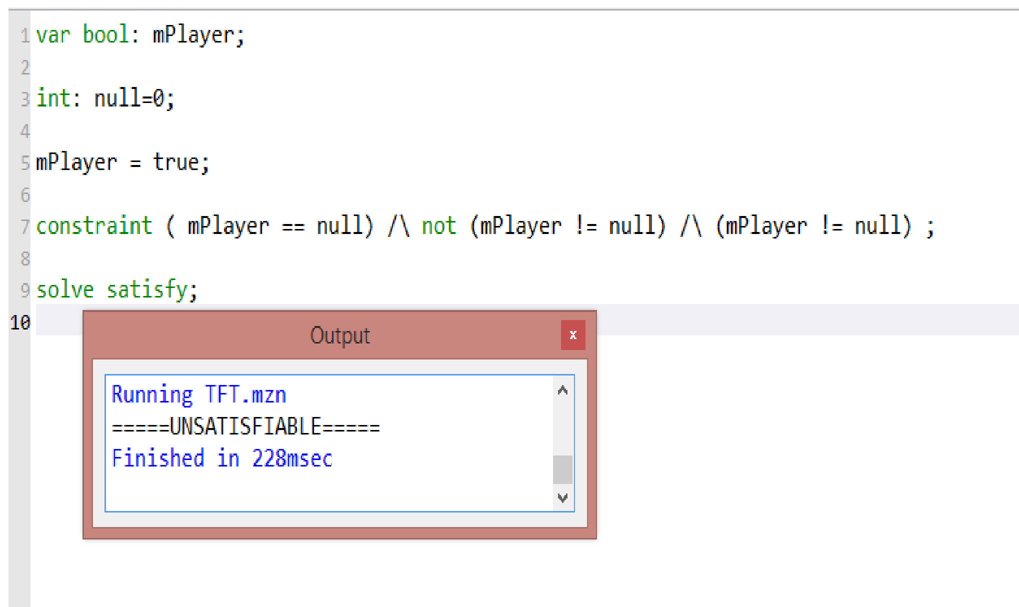

FIGURE 5.8: MiniZinc code and Output of buggy path (TFT)

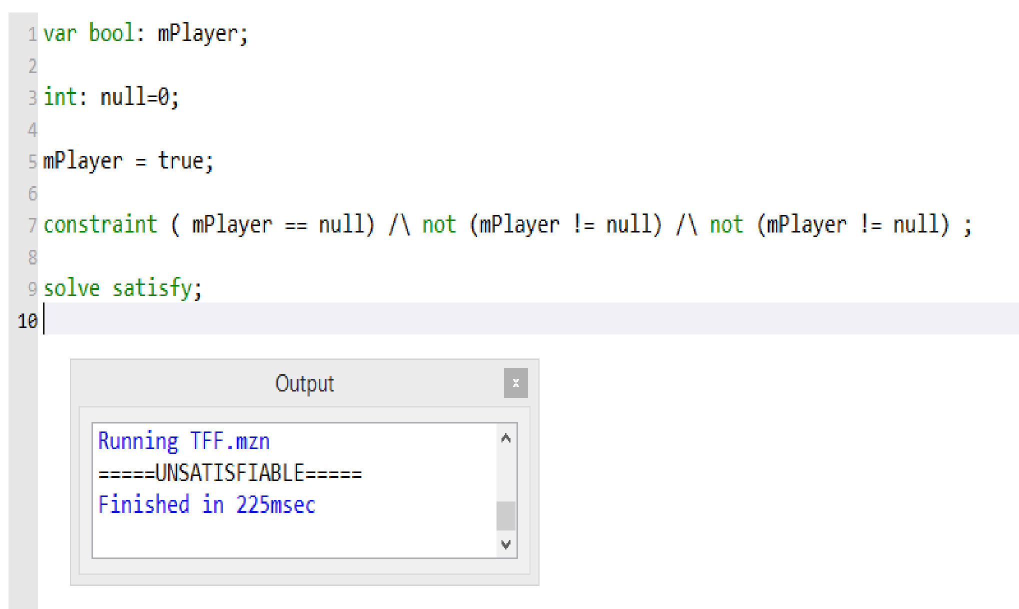

FIGURE 5.9: MiniZinc code and Output of buggy path (TFF)

As shown in the Figure 5.5, the statement presented in line 35 is an infeasible statement as if above conditions at node 28 (28, 29) and at node 32 (32, 33) are executed then there will be no possibility for the given condition at line 35 to be false at all. Thus, every path that has the false implementation of line 35 will be considered an infeasible path. Furthermore, if condition at line 28 is true, media player (MP) will be created, then next condition 32 (32, 33) will be true because it will indicate that media player is not null after being created, therefore if that

condition (i.e. condition at node 32) is false it will also indicate that every path that has the line 32 false implementation will be considered an infeasible path. So, it is identified that three paths were detected as buggy paths and all of them are infeasible paths as detected by our approach.

Another application named ”Fbreader” source code (Figure 5.10), CFG (Figure 5.11), test paths (Table 5.5) and path conditions (Table 5.6) are presented below. Moreover, the MiniZinc code and output for the path conditions of buggy paths are presented in Figure 5.12, Figure 5.13 and Figure 5.14 respectively. The resource kind is Wakelock (WL) that is acquired at line 15 and released at line 25. If the conditions at line 13 will be true then next statements will be executed and wakelock will be acquired otherwise it will skip the next statements and check for the next conditions. If condition at line 21 is true it will indicate that wakelock is null, and if last condition at line 24 is true wakelock will be released.

Moreover, if wakelocks are not used properly this will lead to high consumption of energy.

```
12        public final void createWakeLock() {
13            if (myWakeLockToCreate) {
14                myWakeLockToCreate = false;
15                myWakeLock.acquire();}
16            if (statemyStartTimer) {
17                myFBReaderApp.startTimer();
18                statemyStartTimer = false;}
19            }
20        private final void switchWakeLock(boolean on) {
21            if (myWakeLock == null) {
22                myWakeLockToCreate = true;}
23
24            else { if (myWakeLock != null)
25                {myWakeLock.release();
26                myWakeLock = null; }}
27                    }
```

Figure 5.10: Source code of Fbreader Application

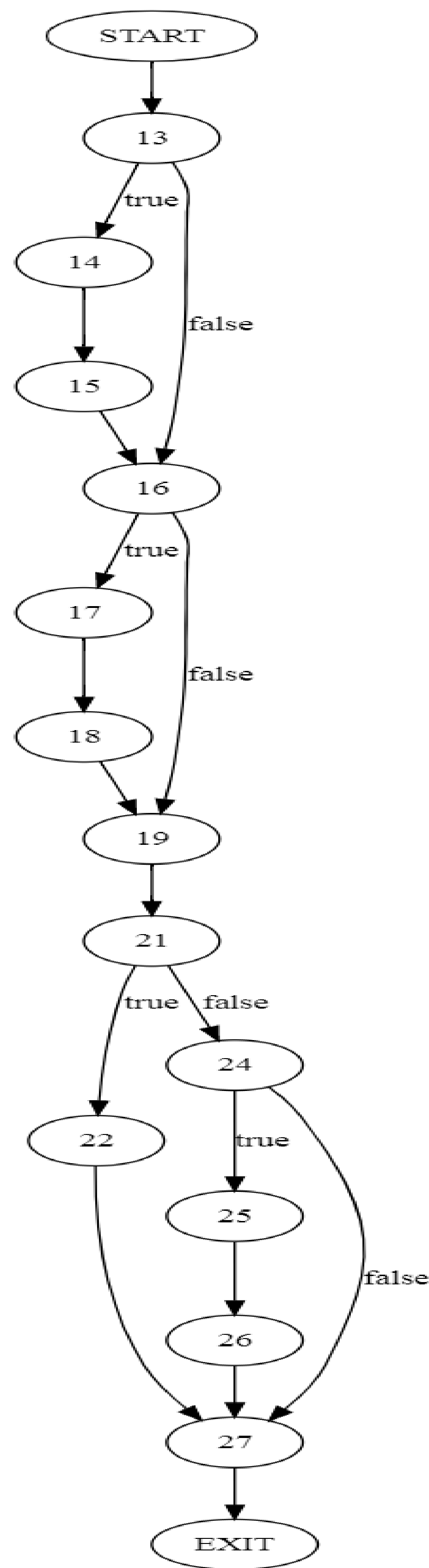

FIGURE 5.11: CFG of Fbreader Application

TABLE 5.5: Test paths, Buggy paths and Resource details

No.	**Test Paths**	**Resource (WakeLock) details**	**Buggy paths**
1.	[Start,13,14,15,16,17,18,19,21,22,27, Exit]	WakeLock is opened but not closed. (TTT)	✓
2.	[Start,13,14,15,16,19,21,22,27, Exit]	WakeLock is opened but not closed. (TFT)	✓
3.	[Start,13,14,15,16,17,18,19,21,24,27, Exit]	WakeLock is opened but not closed. (TTFF)	✓
4.	[Start,13,14,15,16,19,21,24,25,26,27, Exit]	WakeLock is opened and closed. (TFFT)	
5.	[Start,13,16,17,18,19,21,22,27, Exit]	WakeLock is neither opened nor closed. (FTT)	
6.	[Start,13,14,15,16,19,21,24,25,26,27, Exit]	WakeLock is opened and closed. (TTFT)	

TABLE 5.6: Path Conditions for buggy paths

No.	**Buggy paths**	**Path Conditions**
1.	TTT	(myWakeLock) /\ (statemyStartTimer) /\ (myWakeLock== null)
2.	TFT	(myWakeLock) /\ ¬ (statemyStartTimer) /\ (myWakeLock == null)
3.	TTFF	(myWakeLock) /\ (statemyStartTimer) /\ ¬(myWakeLock == null) /\ ¬(myWakeLock != null)

```
var bool: statemyStartTimer;
var bool: myWakeLock;
int: null=0;

myWakeLock = true;

statemyStartTimer = true;

constraint (( myWakeLock) /\  (statemyStartTimer) /\ (myWakeLock == null));

solve satisfy;

```

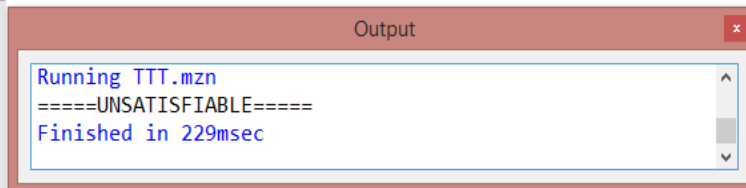

FIGURE 5.12: MiniZinc code and Output of buggy path (TTT)

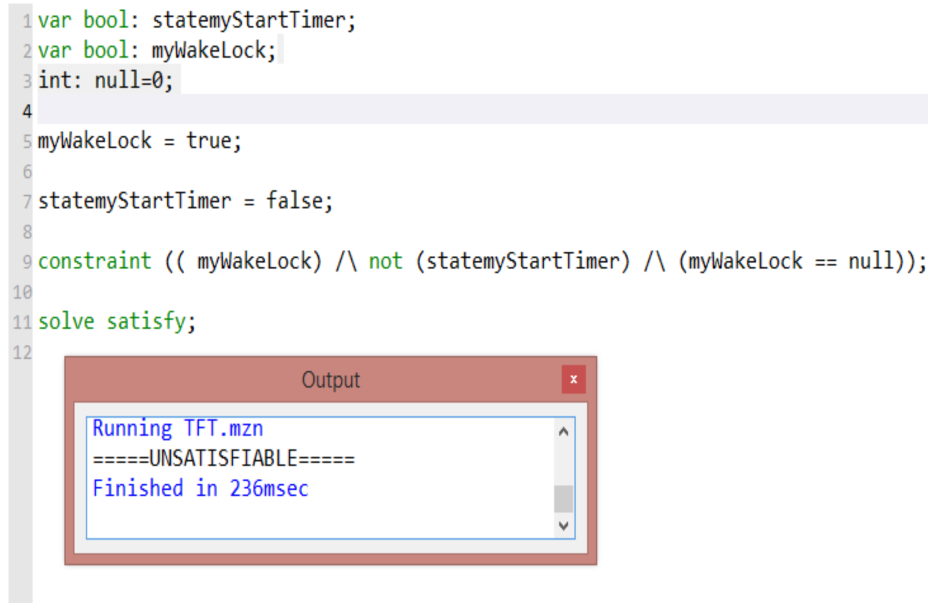

FIGURE 5.13: MiniZinc code and Output of buggy path (TFT)

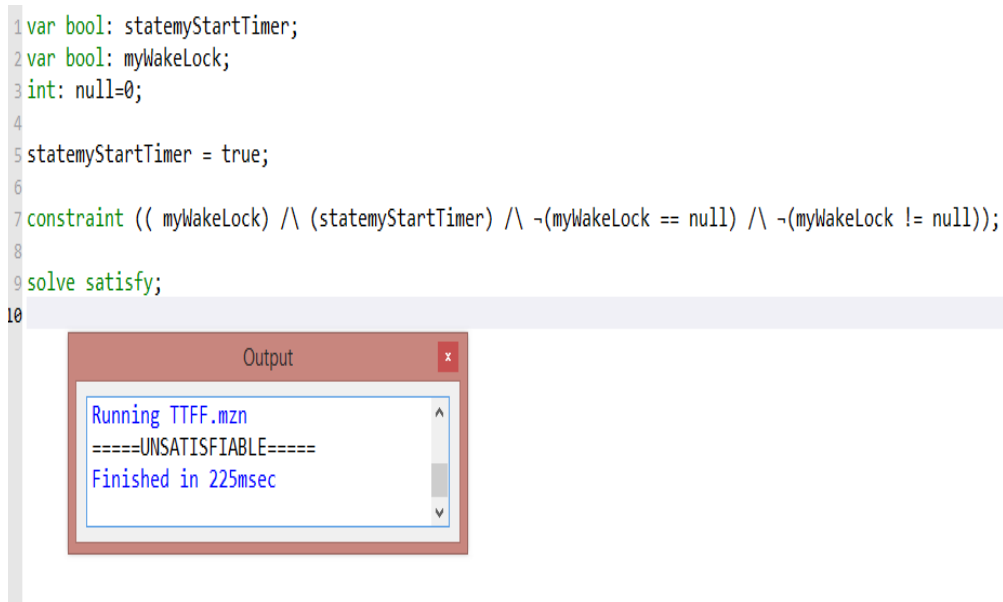

FIGURE 5.14: MiniZinc code and Output of buggy path (TTFF)

As shown in the Figure 5.10, the statement presented at line 21 is an infeasible statement as if above conditions at node 13 (13, 14) is executed then there will be no possibility for the given condition at line 21 to be true at all. Meanwhile, the wakelock is acquired at line 15, and wakelock is being checked that it is null or not at line 21. If wakelock is acquired it cannot be null. Thus, every path that has the true implementation of line 35 will be considered an infeasible path if above condition at node 13 (13, 14) having true implementation. So, it is identified that

three paths were detected as buggy paths and all of them are infeasible paths as detected by our approach.

Furthermore, Table 5.7 summarizes our findings of all selected Android applications (for evaluation of our approach), where O, U, C are the three situations of energy bugs (specified in Table 1.2), where O denotes that an opened resource which is neither used nor closed, U denotes an opened resource which is used but not closed at the end and C denotes that acquired resource is closed eventually without being used.

All the three situations of energy bugs due to the resource leakage are considered in this study. However, there are only open and unclosed-ness (close) behaviours but no use (unused-ness) behaviour for WakeLock. Therefore, for the detection of energy bugs in those application where WL is used only open and close behaviours are considered.

T denotes the total reported energy bugs, FP denotes the false positives that are generated from the proposed approach, R denotes the resource kind that causes the energy leakage due to different type of bugs (i.e. resource leakage and wakelock bugs), and HC, WL, MP represent HTTP connection, WakeLock and Media Player for short respectively. LOC represents the line of codes in each application and numbers of classes in each application are also listed. Besides, description of each application is also mentioned that define the purpose of each application.

Furthermore, several real world Android applications are selected, proposed approach is applied on each application and results are generated. For the purpose of evaluation of proposed approach real world Android applications are selected from the data set of previous approach [21] and some other applications are also selected. Moreover, all there situations of energy bugs are considered for the identification of energy bugs in Android applications.

Results that are generated from the proposed approach and comparison of both approaches (previous [21] and proposed approach) are conferred in the following sections.

Table 5.7: Results for Selected Applications

Apps	Applications Description	O	U	C	T	FP	R	LOC	No. of classes
Adbwireless	An application to connect an Android phone over wifi and even if the phone is not rooted it works.	-	-	-	-	-	WL	781	7
Smarterwifi	An application that manages device Wi-Fi connection. Wi-Fi is only enabled on that location where you previously used Wi-Fi, increasing battery life, security and privacy.	-	-	-	-	-	WL	893	4
Fbreader	An application "favorite book reader" for eBook reading. Where main ebook formats include RTF, doc, HTML etc.	-	-	-	-	-	WL	53,730	69

Apps	Applications Description	O	U	C	T	FP	R	LOC	No. of classes
BabyMonitor	An application to monitor the baby by allowing two Android devices to act as baby monitor.	0	1	0	1	1	MP	731	4
Smspopup	An application that interrupts incoming messages and displays them in a popup.	-	-	-	-	-	MP	8697	40
Tether	App to read any text, pdf, websites. It changes the accent on voice and has ability to have playback info of browser.	-	-	-	-	-	HC	9570	12
Ttrssreader	Voice converter application and provides easy typing and speak options.	1	-	-	1	0	WL	2718	7
BabbleSink	An application that helps to locate the lost phones.	1	-	-	1	0	WL	389	7

Apps	**Applications Description**	**O**	**U**	**C**	**T**	**FP**	**R**	**LOC**	**No. of classes**
Hydromemo	An application which tries to help drink enough water.	1	0	0	1	0	MP	657	11
Betterwifi	Control wifi state and optimizes batter life.	1	-	-	1	0	WL	426	5
Coolreader	An application for reading books in different formats such as fb2, txt, doc, rtf, html, pdf and pml.	2	1	-	3	0	HC	5326	48
AppAlarm	An application to turn any app into an Alarm Clock. To schedule any app for anytime.	1	-	-	1	0	WL	925	10
Pedometer	An application for recoding walking steps / Step Counter app.	1	-	-	1	0	WL	1801	17
Total	-	8	2	-	10	1	-	86644	241

The results show that among these 13 applications, our proposed approach reports that five applications have energy leaks caused by WakeLock; and two applications have energy leaks caused by Media Player and one for HTTP connection. Moreover, the main reason of bugs is that programmers are prone to forget to close/ release the resource for every exit. Different conditions for bugs such as unclosed-ness (i.e. O and U) and unused-ness (i.e. O and C) which are mentioned earlier in Table 1.2.

So, from the results, we can also see that lots of bugs are due to the unclosed-ness (i.e., O and U). Such as, the application Hydromemo creates a MP but does not release it finally, while the application Betterwifionoff creates two WakeLocks (i.e., one for wifi and the other for screen) but only releases the one for screen. Another application BabbleSink creates a WL but does not released it finally, while application Ttrssreader, AppAlarm and Pedometer creates a WL but does not release it finally.

Moreover, there are also some bugs due to the (O). This is mainly because programmers are likely to open the unnecessary resource without considering it is in need or not and doesn't close the resource at the end. Such as, the application coolreader open the Http connection (HC) but does not release it, while in other case HC was opened, and used but not released finally (Resource leak bug). Moreover, in the application BabyMonitor, Media player (MP) is created but the callback function onCompletion, which will release the Media Player(MP), is invoked when the media player finishes the playing. Our approach is not able to identify this callback relation because it is unable to handle a procedure or function in a program that waits for an event to occur termed event listener, therefore false positives are generated.

Furthermore, the graphical representation of results is listed below in Figure 5.15, which shows the different situations of energy bugs (Open, unclosed-ness, unused-ness) and FPs at x-axis and no. of identified bugs according to every application is plotted at y-axis.

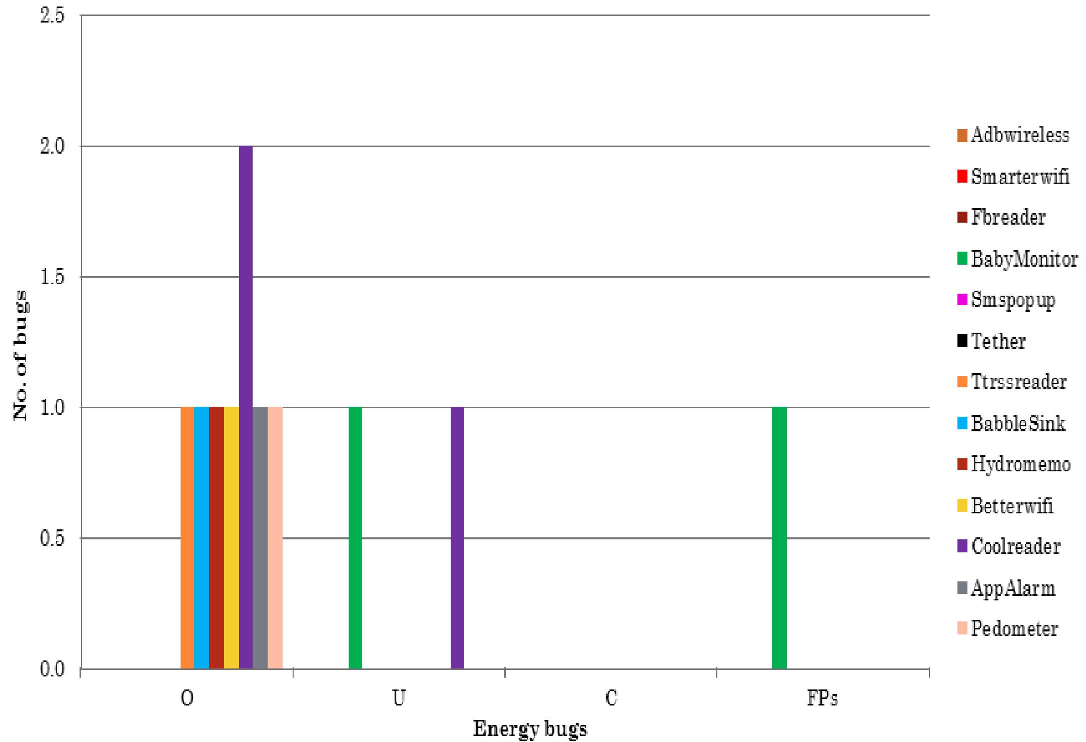

FIGURE 5.15: Results of Selected Applications

The proposed approach results shows that eight bugs are due to (O) which means resource is opened but neither used nor closed, and two of them are indicating bugs due to (U) which means resource is opened, used but not closed. One bug that is generated due to the (U) is FP.

5.2 Comparison

We have compared our proposed technique with the existing technique proposed by Xu et al. [21]. Furthermore, we have implemented the existing technique and generated the path conditions for better identification of energy bugs as well as false positives. Moreover, we have used same dataset that are used by [21] for comparison.

Furthermore, Table 5.8 represents comparison of all the selected Android applications, where results of the previous approach [21] are compared with the proposed approach.

Table 5.8: Comparison of Selected Applications

Results of Previous Approach[21]						**Results of Proposed Approach**						
Apps	**O**	**U**	**C**	**T**	**R**	**Apps**	**O**	**U**	**C**	**T**	**FP**	**R**
Adbwireless	1	-	-	1	WL	Adbwireless	-	-	-	-	-	WL
Smarterwifi	1	-	-	1	WL	Smarterwifi	-	-	-	-	-	WL
Fbreader	1	-	-	1	WL	Fbreader	-	-	-	-	-	WL
BabyMonitor	0	1	0	1	MP	BabyMonitor	0	1	0	1	1	MP
Smspopup	1	0	0	1	MP	Smspopup	-	-	-	-	-	MP
Tether	1	1	0	2	HC	Tether	-	-	-	-	-	HC
Ttrssreader	1	-	-	1	WL	Ttrssreader	1	-	-	1	0	WL
BabbleSink	1	-	-	1	WL	BabbleSink	1	-	-	1	0	WL
Hydromemo	1	0	0	1	MP	Hydromemo	1	0	0	1	0	MP
Betterwifi	1	-	-	1	WL	Betterwifi	1	-	-	1	0	WL
Coolreader	2	1	-	3	HC	Coolreader	2	1	-	3	0	HC
Total	11	3	-	14	-	Total	6	2	-	8	1	-

From the above table, it is concluded that for all the applications, the rate of false positives from our proposed approach is lower than the existing approach [21]. The reason is that our technique uses path conditions to identify those bugs that were identified as real energy bugs. Previous approach [21] identified the energy bugs without evaluating the paths that either identified buggy paths are indicating real energy bugs or not. Static analysis generates the false positives because sometimes those paths are also analysed that can never be executed. If situation of energy bugs exist in any path that can never be executed (i.e. infeasible paths), that path is also considered as energy bugs from static analysis. However, after identification of buggy path we have created path conditions by using symbolic execution. Further, infeasible paths or real energy bugs are identified due to evaluation of path conditions by constraint solver. Therefore, we have overcome the number of false positives. Secondly, stronger graph coverage criterion named Edge coverage criteria is used to cover all the paths effectively. Moreover, graphical representation of the comparison is presented in the Figure 5.16, where at x-axis the true positives (TPs) and false positives that were identified from previous approach [21] and from proposed approach. Likewise, at y-axis, total no. of energy bugs including false positives and TPs were 14 and 7 of them were false positives and 7 of them were TPs from previous approach and from our approach 7 TPs and one energy bugs is turn into FPs.

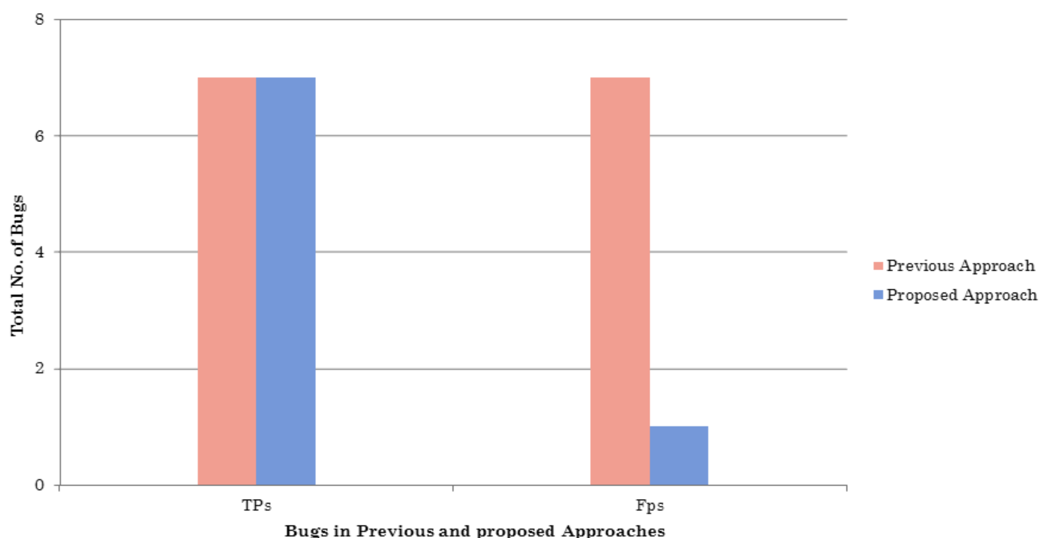

FIGURE 5.16: Comparison of Total Energy Bugs

Furthermore, overall results (as shown in Table.5.7) show that among 13 applications we have identified the false positives as well as real energy bugs. The existing approach was detecting energy bugs in different resources. But there is an issue of generation of false positives from their approach. To overcome this problem, we have enhanced their approach by adding stronger graph coverage criterion. Moreover, the path conditions were not considered, we have also considered the path conditions to identify that whether the bugs that were identified by them were real energy bugs or false positives.

Researchers [21] discussed that they found some false positives through the manual analysis on the source codes of those applications where energy bugs were reported. Different reasons of false positives were also discussed and the main reason was that they have not considered the path conditions and one path may lead to different states for the same resource, hence false positives can be generated.

From the above discussion, it is concluded that the proposed energy bugs detection technique is performing better than the existing energy bugs detection techniques.

Likewise, some of the existing static analysis studies show that from their approach false positives were generated. Such as, study of Pathak et al. [32] show that 55 no-sleep bugs with 13 false positives were found in 86 Apks. Another study of Guo et al. [33] revealed, eleven false positives from their results. Jiang et al. [38] conducted experiments on 64 real APK files and detected that 52 of them have resource leaks, 8 of them were false positives.

Besides, the previous technique [21] has some energy bugs that are false positives, our proposed technique is detecting those bugs that are false positives and it is also minimizing false positives to much extent. In addition, proposed approach is more useful and it is quite less costly due to not considering all the test paths for the path conditions but just the buggy paths.

In the following section mapping study is presented.

5.3 Mapping Study

Table 5.9: Mapping Study

No.	Objectives	Fulfillment of Objectives
1.	Eliminate the false positives to overcome the existing approaches limitations.	Experimental results show that proposed approach is able to overcome the number of FPs. The FPs are reduced but not eliminated. FPs are generated in some cases because the proposed approach is not able to catch the call back relation at the current stage.
2.	All three situations of energy bugs (i.e. open, unclosed-ness and unusedness) are considered for the detection of energy bugs.	Proposed approach is able to cover all the situations of energy bugs for energy bugs detection. In short, we have achieved this objective completely.

As shown in Table 5.9, main objective of this study is to eliminate the false positives because existing static analysis approaches generate more number of false positives. As we know that false positives reduce the quality of results. Researchers [42] stated that generation of false positives is the main problem of static analysis that must need to be considered. Moreover, Zhang et al. [50] stated that some paths are infeasible that is well-known problem with static analysis. Therefore, to decide the feasibility of paths, we need to solve a set of constraints. So, to detect the infeasible paths and to eliminate the false positives we have proposed the static analysis based approach. Proposed approach includes the symbolic execution for creation of path conditions that are evaluated on constraint solver. Resultantly, we identified all the infeasible paths or false positives.

However, due to major limitation of static analysis (false positives generation) our approach is not eliminating the false positives but we have overcome the number of false positives. In other words, false positives that are generated due to static analysis are reduced and we have almost achieved our objective as the proposed

approach minimizes the false positives in much context but not eliminating all the false positives.

Existing static energy bugs detection techniques consider only the second situation of energy bugs named “unclosed-ness”. Due to not considering all the situations of energy bugs (i.e. open, unclosed-ness and unused-ness) there is a possibility to miss some energy bugs. Therefore, the proposed approach considers all the situations of energy bugs for the detection of energy bugs.

Chapter 6

Conclusion and Future Work

In this dissertation, we have discussed how to effectively detect the energy bugs in Android applications. In particular, we have focused on one of the main causes of high battery consumption i.e. energy bugs. Smartphone emergence has increased from the past few years. It has surpassed the desktop machines and became the most popular and useful due to its multiple features such as large screen, GPS and camera. The faster growth in specifications of Android phones allows the developers to release the loads of applications every day. However, Android battery consumption is still high due to complex applications. Moreover, to overcome the battery issues a lot of approaches have been proposed for the energy bugs detection in Android applications. After critical analysis of literature, we have observed that these approaches have three categories that are 1) Static analysis approaches, 2) Dynamic analysis approaches and 3) Hybrid approaches. Static approaches generate more number of false positives (FPs) due to not executing the code. Therefore, they are not able to completely detect all the energy bugs. However, dynamic and hybrid approaches are more costly due to execution of test cases and high execution time respectively.

We argue that for better detection of energy bugs FPs needs to be overcome. In existing state-of-the art, researchers have picked different methods of detecting the energy bugs but still static approaches generate more number of FPs. The objective of this study is to consider all the possible situations of energy bugs (i.e.

open, unclosed-ness and unused-ness) and to overcome the FPs by identification of infeasible paths that leads to FPs.

In this study, we have enhanced the state-taint analysis, guided by resource usage protocols, for better detection of energy bugs. Furthermore, path conditions are created for the bug prone paths and constraint solver is used to identify the real energy bugs. To demonstrate the viability of the approach, several experiments on real Android applications are done to detect the energy bugs, where the applications are collected from well-known repositories i.e. Git-hub and Google code. Using our approach, we were able to uncover real energy bugs and FPs in real-world applications. We have compared our results with state-of-the-art approach presented by Xu et al. [21]. Our study resulted in real energy bugs and infeasible paths which results in FPs and experimental results show that our approach is helpful and suitable in practice to detect and overcome the FPs. In addition, this work aims to detect the energy bugs in Android applications with lower number of false positives.

The overall findings of this study are, 1) better detection of energy bugs 2) detect the FPs and the last one 3) the proposed approach reduces the FPs rate in the static analysis approaches.

Moreover, following are the answers of our research questions portrayed in Chapter 1, which we have identified after doing literature review, investigation and experimentation.

RQ 1: What are the shortcomings of existing static energy bugs detection techniques?

The critical analysis of already proposed static approaches in the field of energy bugs are investigated thoroughly and some gaps are identified. These approaches consider only the second situation of the energy bugs named “unclosed-ness”. Moreover, these approaches generate more number of false positives. The reason behind generation of false positives is that static approaches perform without executing the code. It cannot evaluate the conditions so some of the paths can be

added as buggy paths which are not real energy bugs. Therefore, false positives can be generated due to static analysis.

RQ 2: How to overcome the shortcomings of existing static energy bugs detection techniques?

Static approaches perform without execution of code. Therefore, these approaches generate more number of false positives. To overcome this limitation, our technique identifies the buggy paths, and by using the symbolic execution creates path conditions for buggy paths. Moreover, constraint solver is used for evaluating the path conditions to determine the infeasible paths or false positives. This helps in detect and overcome the false positives in static approaches. Besides, the proposed approach considers three situations of energy bugs including open, unclosed-ness and unused-ness.

RQ 3: Does the proposed approach overcome the false positives when compared with existing techniques?

We have proposed energy bugs detection technique which detect the energy bugs in Android apps and overcome the false positives in static analysis. Proposed technique identifies the buggy paths and after that path conditions are created for the identification of real energy bugs. We have performed experiments to detect the energy bugs on different real world Android applications for determining the effectiveness of the proposed technique over existing static analysis techniques. We have enhanced the existing state-taint analysis by adding the stronger graph coverage criteria and path conditions are added that are further evaluated for the identification of infeasible paths that leads to false positives. Previous approaches generate more number of false positives and after experiments results show that our technique overcomes this limitation by using stronger graph coverage criterion and path conditions that are evaluated for the identification of false positives. For comparison, previous approach [21] is compared with proposed approach and final results show that our technique performs better than the existing techniques. Moreover, proposed technique is able to effectively detect the real energy bugs and false positives.

6.1 Future Work

We have identified some of probable directions for future research in this area. In the future, our approach can be extended by considering the callback relation and exceptions as different bugs can arise due to exceptions. In this study, we have conducted experiments on several real Android applications and test datasets from state-taint analysis. In future, some additional case studies for comparison and evaluation can be used. Furthermore, with more investigation, this study can provide guidelines to the developers in order to help testing their applications and make sure they are free of energy bugs.

Bibliography

[1] L. Cruz, R. Abreu, and J.-N. Rouvignac, "Leafactor: Improving energy efficiency of android apps via automatic refactoring," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft).* IEEE, 2017, pp. 205–206.

[2] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE communications surveys & tutorials*, vol. 17, no. 2, pp. 998–1022, 2014.

[3] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, "Towards verifying android apps for the absence of no-sleep energy bugs," in *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*, 2012.

[4] X. Gao, D. Liu, D. Liu, H. Wang, and A. Stavrou, "E-android: A new energy profiling tool for smartphones," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS).* IEEE, 2017, pp. 492–502.

[5] A. Degu, "Android application memory and energy performance: Systematic literature review," *IOSR J. Comput. Eng.*, vol. 21, no. 3, pp. 20–32, 2019.

[6] 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.

[7] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 92–101.

[8] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011, pp. 1–6.

[9] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker, "edoctor: Automatically diagnosing abnormal battery drain issues on smartphones," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 57–70.

[10] 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.

[11] A. Schuler and G. Anderst-Kotsis, "Towards a framework for detecting energy drain in mobile applications: an architecture overview," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018, pp. 144–149.

[12] J. Zhang, A. Musa, and W. Le, "A comparison of energy bugs for smartphone platforms," in *2013 1st international workshop on the engineering of mobile-enabled systems (MOBS)*. IEEE, 2013, pp. 25–30.

[13] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 29–42.

[14] 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.

[15] 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.

[16] Z. Xu, D. Fan, and S. Qin, "State-taint analysis for detecting resource bugs," in *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2016, pp. 168–175.

[17] D. Li, S. Hao, J. Gui, and W. G. Halfond, "An empirical study of the energy consumption of android applications," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 121–130.

[18] A. W. Marashdih and Z. F. Zaaba, "Infeasible paths in static analysis: Problems and challenges," in *AIP Conference Proceedings*, vol. 2016, no. 1. AIP Publishing LLC, 2018, p. 020079.

[19] M. Papadakis and N. Malevris, "A symbolic execution tool based on the elimination of infeasible paths," in *2010 Fifth International Conference on Software Engineering Advances*. IEEE, 2010, pp. 435–440.

[20] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International journal on software tools for technology transfer*, vol. 11, no. 4, p. 339, 2009.

[21] Z. Xu, C. Wen, and S. Qin, "State-taint analysis for detecting resource bugs," *Science of Computer Programming*, vol. 162, pp. 93–109, 2018.

[22] 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.

[23] 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.

[24] K. Kim and H. Cha, “Wakescope: runtime wakelock anomaly management scheme for android platform,” in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. IEEE, 2013, pp. 1–10.

[25] A. Ferrari, D. Gallucci, D. Puccinelli, and S. Giordano, “Detecting energy leaks in android app with poem,” in *2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. IEEE, 2015, pp. 421–426.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] C. Zhu, Z. Zhu, Y. Xie, W. Jiang, and G. Zhang, “Evaluation of machine learning approaches for android energy bugs detection with revision commits,” *IEEE Access*, vol. 7, pp. 85 241–85 252, 2019.

[32] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, “What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone

apps," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 2012, pp. 267–280.

[33] 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.

[34] 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.

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

[36] 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.

[37] C. H. P. Kim, D. Kroening, and M. Kwiatkowska, "Static program analysis for identifying energy bugs in graphics-intensive mobile apps," in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2016, pp. 115–124.

[38] H. Jiang, H. Yang, S. Qin, Z. Su, J. Zhang, and J. Yan, "Detecting energy bugs in android apps using static analysis," in *International Conference on Formal Engineering Methods*. Springer, 2017, pp. 192–208.

[39] S. Hall, S. Nataraj, and D.-K. Kim, "Detecting no-sleep energy bugs using reference counted variables," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, 2018, pp. 161–165.

[40] S. Hall, S. Nataraj, and D.-K. Kim, "Detecting no-sleep energy bugs using reference counted variables," in *Proceedings of the 43rd Annual Computer Software and Applications Conference*, 2019, pp. 940–941.

[41] W. Song, J. Zhang, and J. Huang, "Servdroid: detecting service usage inefficiencies in android applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 362–373.

[42] A. W. Marashdih, Z. F. Zaaba, and S. M. Almufti, "The problems and challenges of infeasible paths in static analysis," *International Journal of Engineering & Technology*, vol. 7, no. 4.19, pp. 412–417, 2018.

[43] S. Ding and H. B. K. Tan, "Detection of infeasible paths: Approaches and challenges," in *International Conference on Evaluation of Novel Approaches to Software Engineering*. Springer, 2012, pp. 64–78.

[44] D. Gong and X. Yao, "Automatic detection of infeasible paths in software testing," *IET software*, vol. 4, no. 5, pp. 361–370, 2010.

[45] B. Barhoush and I. Alsmadi, "Infeasible paths detection using static analysis," *The Research Bulletin of Jordan ACM*, vol. 2, no. 3, pp. 120–126, 2013.

[46] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[47] R. Aïssat, F. Voisin, and B. Wolff, "Infeasible paths elimination by symbolic execution techniques," in *International Conference on Interactive Theorem Proving*. Springer, 2016, pp. 36–51.

[48] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1066–1071.

[49] V. G. Riyad Parvez, Paul A.S. Ward, "Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries," in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, 2016, pp. 116–127.

[50] J. Zhang and X. Wang, "A constraint solver and its application to path feasibility analysis," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 02, pp. 139–156, 2001.

[51] S. Alekseev, V. Dhanraj, S. Reschke, and P. Palaga, "Tools for control flow analysis of java code," in *Proceedings of the 16th IASTED International Conference on Software Engineering and Applications*, 2012.

Appendix

CFG is created for example code shown in Figure 4.1. After that, CFGs is exported to the XML format which is shown below. Moreover, CFG is exported the graph in DOT format for the visualization of graph in GraphViz.

XML of the CFG (shown in Figure 4.2)

<GraphXML>

<graph version=”1.0” vendor=”www.drgarbage.com”
id=”IncomingSms.findMyLocation.src.graph”>

<node name=”2”>

<label>76</label>

</node>

<node name=”3”>

<label>77</label>

</node>

<node name=”4”>

<label>78</label>

</node>

<node name=”5”>

<label>79</label>

</node>

<node name=”6”>

<label>80</label>

</node>

<node name=”7”>

<label>81</label>

</node>

<node name=”8”>

<label>82</label>

</node>

<node name=”9”>

<label>83</label>

</node>

<node name=”10”>

<label>84</label>

</node>

<node name=”11”>

<label>85</label>

</node>

<node name=”12”>

<label>86</label>

</node>

<node name=”13”>

<label>87</label>

</node>

<node name=”14”>

<label>88</label>

</node>

<node name=”15”>

<label>89</label>

</node>

<node name=”16”>

<label>90</label>

</node>

<node name=”17”>

<label>91</label>

</node>

<node name=”18”>

<label>START</label>

</node>

<node name=”19”>

<label>EXIT</label>

</node>

<edge source=”18” 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>true</label>

</edge>

<edge source=”8” target=”9”>

<label/>

</edge>

<edge source=”9” target=”10”>

<label>true</label>

</edge>

<edge source=”10” target=”11”>

<label/>

</edge>

<edge source=”9” target=”12”>

<label>false</label>

</edge>

<edge source=”11” target=”12”>

<label/>

</edge>

<edge source=”12” target=”13”>

<label>true, false</label>

</edge>

<edge source=”13” target=”14”>

<label/>

</edge>

<edge source=”14” target=”15”>

<label/>

</edge>

<edge source=”7” target=”16”>

<label>false</label>

</edge>

<edge source=”12” target=”17”>

<label>false</label>

</edge>

<edge source=”16” target=”17”>

<label/>

</edge>

<edge source=”15” target=”19”>

<label/>

</edge>

<edge source=”17” target=”19”>

<label/>

</edge>

</graph>

</GraphXML>

CFG is exported the graph in DOT format for the visualization of graph in GraphViz. Dot format for the CFG of example code shown in Figure 4.1, is shown below.

Dot format of the CFG (shown in Figure 4.2)

<!– –> digraph ”IncomingSms.findMyLocation.src.graph” {

graph [label=”IncomingSms.findMyLocation.src.graph”];

2 [label=”76”]

3 [label=”77”]

4 [label=”78”]

5 [label=”79”]

6 [label=”80”]

7 [label=”81”]

8 [label=”82”]

9 [label=”83”]

10 [label=”84”]

11 [label=”85”]

12 [label=”86”]

13 [label=”87”]

14 [label=”88”]

15 [label=”89”]

16 [label=”90”]

17 [label=”91”]

18 [label=”START”]

19 [label=”EXIT”]

18 − > 2 [label=””]

2 − > 3 [label=””]

3 − > 4 [label=””]

4 − > 5 [label=””]

5 − > 6 [label=””]

6 − > 7 [label=””]

7 − > 8 [label=”true”]

8 − > 9 [label=””]

9 − > 10 [label=”true”]

10 − > 11 [label=””]