Analyzing Android Applications for Specifications and Bugs

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Wei Le, Supervisor

Dr. Matthew Fluet, Reader

Dr. Hans-Peter Bischof, Observer

Analyzing Android Applications for Specifications and Bugs

by

Danilo Dominguez, B.E.

THESIS

Presented to the Faculty of the Golisano College of Computer and

Information Sciences

Rochester Institute of Technology

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science

Rochester Institute of Technology

May 2013

Acknowledgments

I wish to thank my advisor Dr. Wei Le for all her support to finish this work and the rest of my committee, Dr Matthew Fluet and Dr. Hans-Peter Bischof, for their consideration on helping me in this thesis. In addition, I would like to thank my family and my fiancee, Dayiris, for their support during the whole time I have been far from home.

Abstract

Analyzing Android Applications for Specifications and Bugs

Danilo Dominguez, M.S.
Rochester Institute of Technology, 2013

Supervisor: Dr. Wei Le

Android has become one of the leader operating systems for smartphones. Moreover, Android has a big community of developers with over 696500 applications on its market. However, given the complexity of the system, bugs are very common on Android applications–such as security vulnerabilities and energy bugs. Given the architecture of Android, current static analysis tools are not suitable for Android applications.

In this thesis, two approaches to analyze Android applications are studied. The first approach is an intra-component analysis that take take in account just the lifecycle of the components to define control flow of the applications. This approach is evaluated applying a specification miner for energy related specifications on 12 applications from the Android market. We were able to mine 91 specifications on all the applications and 41 of them were

validated. For 50% of the applications analyzed, the analysis had less than 40% of false positives specifications. However, for the rest of the applications, the interaction between components was a important factor that increased the false positives.

Therefore, the second approach is an inter-component approach that takes in account both, the lifecycle of components and interaction between components to define the control flow of Android applications. We evaluate the approach checking the percentage of code coverage on 8 applications from the Google market. The results are promising with an average coverage of 67%. In addition, we were able to identify bugs related to violations of constraints regarding inter-component interactions.

Table of Contents

Acknowledgments **iii**

Abstract **iv**

List of Tables **viii**

List of Figures **ix**

Chapter 1. Introduction **1**

Chapter 2. Background and Related Work **5**

2.1 Background 5

2.1.1 Android Overview 5

2.1.2 AndroidManifest, Intent and Intent Filters 9

2.1.3 Interaction between components 10

2.1.4 Energy related API in Android 12

2.1.5 Control Flow Graph 13

2.2 Related Work 13

2.2.1 Security Analysis using Program Analysis for Android . 13

2.2.2 Energy Related Analysis 15

2.2.3 Specification Mining 16

2.2.4 Modeling Event-driven Systems 17

Chapter 3. Mining Energy Specifications (E-Specs) and Energy Bugs (E-Bugs) **18**

3.1 Modeling Intra-Component Control Flow 18

3.2 Specification Miner 19

3.3 Post-Processing 21

3.4 Implementation 21

3.4.1 Getting Intermediate Representation of Android Applications . . . 21
3.4.2 Optimizations . . . 22
3.4.3 Seed Classes from the Android API . . . 23
3.5 Experimental Results . . . 25
3.6 Conclusion . . . 31

Chapter 4. Building a Representation for Analyzing Android Applications **33**
4.1 Motivation . . . 33
4.2 Modeling Inter-Component Control Flow . . . 34
4.3 Modeling Android using Event-driven Finite State Machines . 35
4.3.1 External Signals . . . 36
4.3.2 Internal Signals . . . 36
4.3.3 How External and Internal Signals are treated in our model 37
4.4 Models for Components . . . 38
4.4.1 Modeling Activities . . . 38
4.4.1.1 General Model . . . 41
4.4.2 Modeling Services . . . 44
4.4.2.1 General Model . . . 48
4.4.3 Modeling Broadcast Receivers . . . 51
4.5 Representation . . . 52
4.6 Construction of the AIG . . . 53
4.7 Experimental Results . . . 55
4.8 Conclusion . . . 58

Chapter 5. Conclusions and Future Work **59**

Bibliography **61**

List of Tables

3.1 Benchmarks 26
3.2 General Specification Mining Results 27
3.3 False Positives for the Specifications 30

4.1 Inter-component analysis results 56

List of Figures

2.1 Stack of activities for a process [5] 6
2.2 Activity lifecycle [1] 7
2.3 Service lifecycles [4] 8
2.4 Inter-component method calls 11

3.1 Framework 19
3.2 CFG example for linear marker 23
3.3 Source Code example for linear marker 24
3.4 Performance: No. of classes vs. Analysis time 27
3.5 Source Code 28
3.6 Example of results 29

4.1 Inter-component specification 34
4.2 Activity Launch Task State Machine 39
4.3 startActivity from Activity 39
4.4 Pause and Stop Activity 40
4.5 Restart Activity 40
4.6 Destroy Activity 41
4.7 General Model for Activity 41
4.8 startService when service is not running 45
4.9 startService when service is bound 45
4.10 stopService when service is Started 46
4.11 stopService when service is Started and Bound 46
4.12 bindService when service is not running 46
4.13 bindService when service is started 47
4.14 General Model for Service 48
4.15 BroadcastReceiver Task State Machine 51
4.16 Activity used to build an AIG 54

4.17 Example of an AIG 56
4.18 Sipdroid 5.7 helper function to create intents 57

Chapter 1

Introduction

Android is an operating system developed by Google that targets mobile devices. It has become the most used platform around the world with over 400 million devices activated and another 1 millions activations every day [2]. In addition, the marketplace has over 10 million application purchases since 2012 [11], with over 696500 applications. These numbers have called the attention of hundreds of programmers around the world that see Android application development as a profitable market. Programmers normally use the Android SDK to develop, debug, deploy and launch Android applications directly to the market. This SDK provides an application programming interface (API) that let programmers' code interact with the system using the Java programming language for the development.

Moreover, it is well known that nowadays almost every piece of software uses a third party API. Therefore, one of the main sources of software bugs is the misuse of APIs [45]. In many cases, the misuse of APIs is due to the lack of documentation or incorrect documentation regarding the API implementation. Consequently, programmers can easily make false assumptions or try to find the correct implementation in other sources such as forums

or related websites leading to software bugs. Also, the Android system has an event-driven architecture which uses on a set of components to represent the behavior of applications. Therefore, in order to develop and reason about Android applications, programmers must understand the functionality of each component, how each component interacts with the system, and how different components interact with each other.

Along with all the complexity that programming with an API carries– graphical user interface (GUI), networking, etc.– programming for mobile devices adds more complexity. For instance, power consumption, memory constraints and CPU power constraints are some aspects programmers must take in account at any time during the development of an Android application (these constraints are faced by programmers of embedded systems). Therefore, mobile devices' programmers of Android applications face problems seen by desktop and server computer programmers and problems seen by embedded system programmers.

To illustrate these complexities, let us use power consumption as example. It is well known that one of the main problems in smartphones is their power consumption. This can be due to the complexity of some computations, networking communication, the use of the sensors and also the screen of the device. That is why, systems such as Android have introduced mechanisms to save energy. For instance, Android enforces strict power policies to keep every sensor and CPU off (or in an idle state) unless an application tells the OS to keep the component on [43]. However, the misuse of these mechanisms has

led programmers introduce a new kind of energy bug: no-sleep bugs. These kind of bugs and the majority of the bugs related to programming with Java are inherits by Android applications. Therefore, program analysis tools can be helpful for Android programmers.

However, given the Android's architecture and its API, current Java tools– such as FindBugs [9] and JLint [10]–are not very suitable for Android application. One of the main challenges to analyzed Android applications is their event-driven architecture. In contrast to most of the Java applications, Android applications does not have a single entry point. Moreover, interactions with the environment (user and network and sensor events) make the analysis and development of Android application a very complex task. There are other explicit system calls that can change the control flow in an Android application that tools for analysis of Java applications does not take in account. Therefore, before applying any kind of analysis on Android applications, the Android system must be modeled. Then, this model can be used to generate a more accurate representation of the execution of an Android application and how the control of execution flow.

In this thesis two approaches to model and analyze Android applications were designed and implemented. The first approach is an intra-component analysis that take take in ac- count just the lifecycle of the components to define control flow of the applications. This approach is evaluated applying a specification miner for energy related specifications on 12 applications from the Android market. We were able to mine 91 specifications on all the applications

and 41 of them were validate. For 5040the interaction between components was a important factor that increased the false positives. Therefore, the second approach is an inter-component approach that takes in account both, the life-cycle of components and interaction between components to define the control flow of Android applications. We evaluate the approach checking the percentage of code coverage on 8 applications from the Google market. The results are promising with an average coverage of 67%. In addition, we were able to identify bugs related to violations of constraints regarding inter-component interactions.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Android Overview

Android is an operating system based on the Linux kernel developed by Google, primarily targeting mobile devices. It offers an API that let programmers develop third party applications to run in the phone–with the API programmers can access the different sensors, cameras and other components the devices have. Each application runs on its own Linux user ID with its own virtual machine (VM) [3]. Google developed its own virtual machine (Dalvik VM) which runs different bytecode to the bytecode run by the Oracle Java Virtual Machine.

Android applications are composed by four main components: Activity, Service, BroadcastReceiver and ContentProvider. Activity classes must be subclasses of *android.app.Activity* (and subclasses declared in the Android API), services from the class *android.app.Service* or *android.app.IntentService*, content providers from *android.content.ContentProvider* and broadcast receivers from the class *android.app.BroadcastReceiver*. Following a description of each component is presented.

Activity

An activity provides an user interface that let users interact with an application [3]. In Android each window can be taken as one activity. Moreover, activities can call other activities from the same application and also from other applications. For example, when an user opens a chat application, the application shows a list of friends available for chatting. If the user tabs on the name of the friend, the system will stop the activity that list the friends and start an activity that let the user chat with the particular friend that was selected.

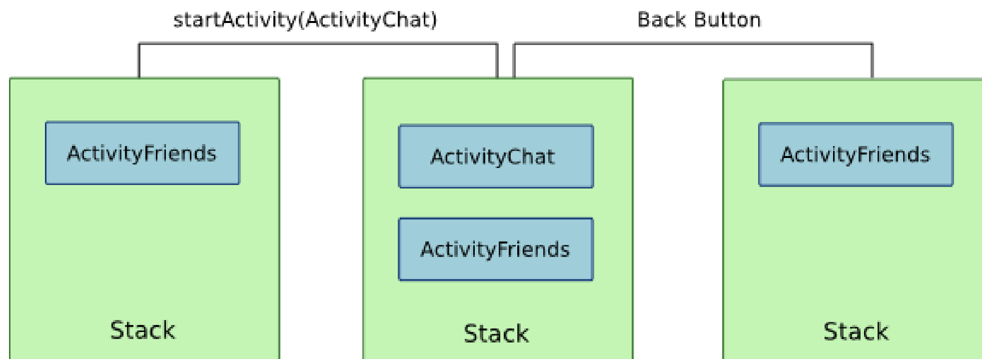


Figure 2.1: Stack of activities for a process [5]

All the activities for a particular process are stored in an stack (in memory) and every time a new activity is started from another activity, the caller activity is stopped, and the new activity is launch and put at the top of the stack. If the user clicks that back button, the system stops the activity at the top of the stack, remove it and restart the activity that is now at the top of the stack–if there are no more activities, the system goes to the home

screen (see figure 2.1).

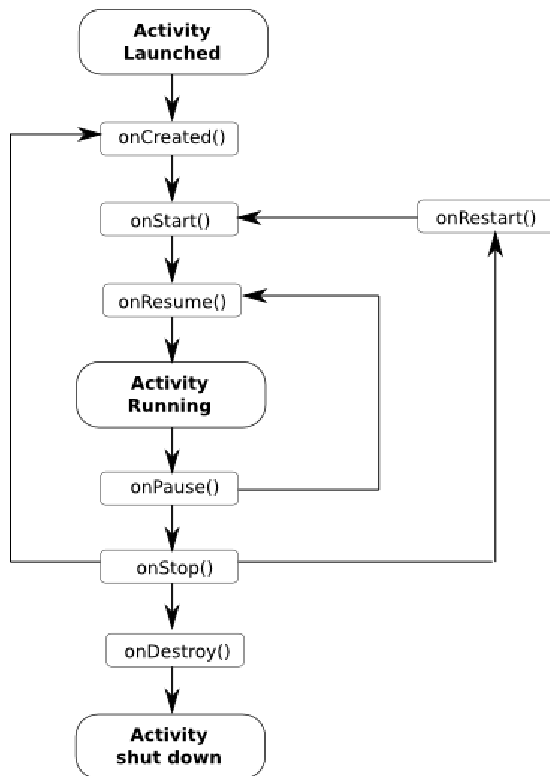
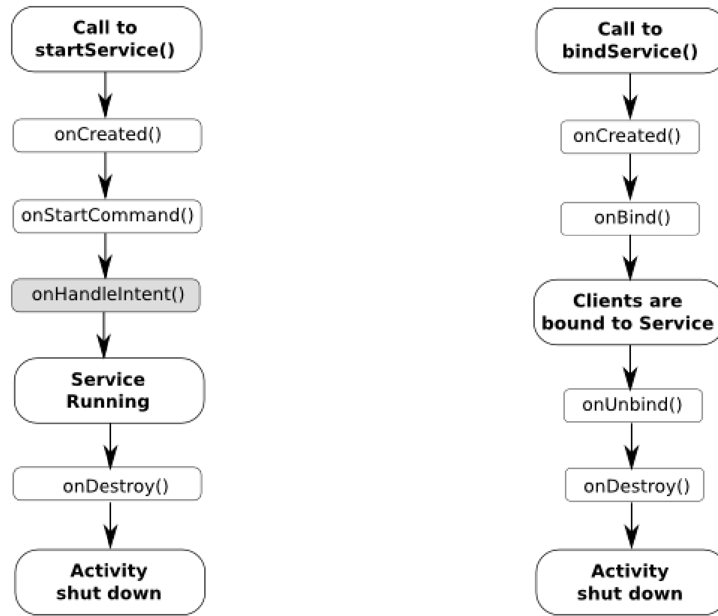


Figure 2.2: Activity lifecycle [1]

When the activity is launched, stopped or restart by the system, the system notifies of these changes using lifecycle callback methods [1]. Figure 2.2 shows the lifecycle for activities and list the activities' callback methods. These lifecycle callback methods are vital for registering and releasing resources the applications need.

Service

In contrast to activities, which normally offer a user interface to interact with



(a) Service lifecycle for startService (b) Service lifecycle for bindService

Figure 2.3: Service lifecycles [4]

the user in the foreground, a service is used for long-running tasks that are executed in the background [3]. Services are normally started by activities, broadcast receivers or other services and they can be called using two different inter-component method calls (each form has a different lifecycle):

- *startService*: if the service is started, it runs in the background without having any dependency to other component. Figure 2.3a shows the life-cycle of a service when it is started. When a service is started it can be stopped by calling the inter-component method call *stopService* or if the service calls the method *stopSelf*. In case the service is an *IntentService*, the callback method *onHandleIntent* is called as well.

- *bindService*: when a service is bound, it is normally called to perform a task that is delegated by the caller component which requires some interaction from the component called. Multiple components and application can bind the same service, but the service will run until there are no more components bound to it [4] (the caller component must call *unbindService* to unbind it). Figure 2.3b shows the lifecycle for services that are bound.

BroadcastReceiver

A broadcast receiver is used to receive notifications mostly from the system but also from applications [3]. When the brodcast receivers gets a notification, the callback method *onReceive* is executed. After this callback finishes, the life the component finishes as well (it is not stored in memory as activities).

ContentProvider

Content providers are used as databases of applications and are normally used if an application wants to share some data to other applications–although there is no any constraint if they are used in the same application which declare it.

2.1.2 AndroidManifest, Intent and Intent Filters

The file where permissions, components and dependencies of libraries are declared is *AndroidManifest.xml*. The programmer can also declare the Java package for the application. Following, some of the most important components of this file are described.

IntentFilter

From the four main components, activities, services and broadcast receivers are called through objects of type *Intent*–this objects describes the components to be called. In the *AndroidManifest.xml* each component declared, can have zero or more intent filters. Intent filters are use to define the capabilities of the components and what kind of intents they can receive. Therefore, a component can have more than one intent filter if the programmer wants the component to be executed by different kind of intents.

When an application uses a inter-component method call–such as *startService* or *sendBroadcast* for broadcast receivers–the description of the target components is stored in an *Intent* object. There are two kind of intents: explicit and implicit intents. Explicit intents has the component's name it targets, whereas implicit intents contains general information of the object(s) it wants to target. The target components are matched by the system based on the intent filters declared by each components across all the applications installed. In a case where more than one activity match the intent sent, the system prompts a list of the activities matched in order to let the user choose which activity to execute. For services and broadcast receivers, there is not any interaction from the user.

2.1.3 Interaction between components

There are different interactions that Android components can have. Normally, activities, services, and broadcast receivers are activated **asyn-**

chronously using inter-component method calls and *Intent* objects.

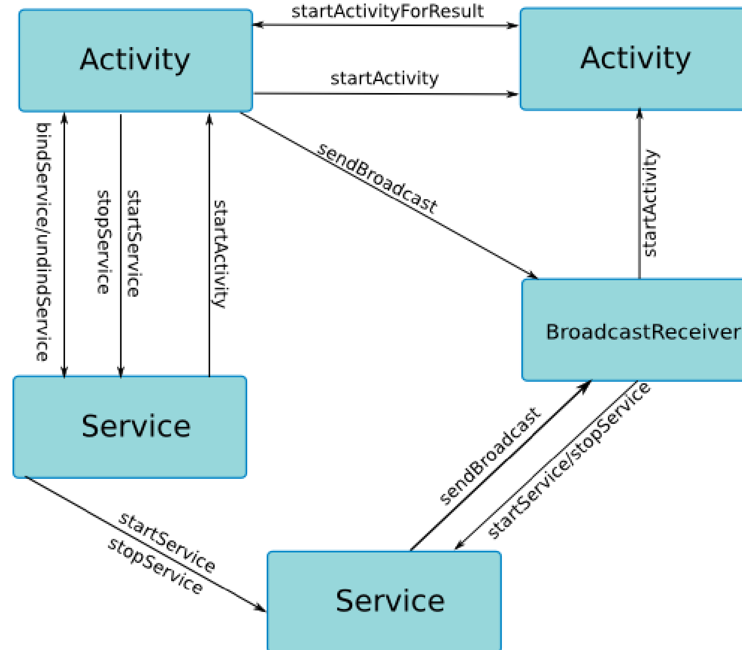


Figure 2.4: Inter-component method calls

Android provides specific inter-component method calls that permit the interaction between different components. Figure 2.4 shows these method calls and what particular components can use them. Following, we list the inter-component method calls categorized by component:

- Activity: startActivity, startActivities, startActivityForResult.
- Service: startService, stopService, bindService, unbindService.
- BroadcastReceiver: sendBroadcast, sendBroadcastAsUser, sendSticky-Broadcast, sendStickyBroadcastAsUser, sendOrderedBroadcast, sendOrdered-

BroadcastAsUser, sendStickyOrderedBroadcast, sendStickyOrderedBroadcastAsUser.

All these system calls are handled by the class *android.app.ActivityThread*. The implementation of the class *android.app.ActivityThread* uses a queue task within a loop to handle all inter-component method calls and other actions posted by the system. For instance, when the application starts a service, the service is not started immediately after the call (*startService(Intent intent)*). Contrary, it is posted to the queue task and the actions can be performed in the next iterations of the loop. Therefore, if we want to create a control flow graph that is closer to the real execution of the system, we have to model how the class *android.app.ActivityThread* executes the different tasks.

2.1.4 Energy related API in Android

As it was mentioned, Android enforces strict energy policies to reduce power consumption. Normally device components such as CPU, the Wifi antenna and other sensors go to an idle state when the phone goes to sleep. Nevertheless, there are some mechanisms that let third party applications maintain awake these components even when the phone is sleeping. One of these mechanisms is called wake locks. For instance, the class *android.os.PowerManager.WakeLock* can be used to maintain the CPU and even the screen on when they are supposed to be off. This mechanism is very important for applications such as music players that keep playing music even when the phone is supposed to be sleeping. When a third party application

wants stop a component to go to an idle state, the application must acquire a *wake lock* on the device component. Once the application is done with its tasks, it must release the lock on the device component.

However, given the concurrent, event-driven Android's architecture, programmers make mistakes on releasing the locks for all the possible paths the application can have. Therefore, the device component will not go to idle state, causing a great amount of energy consumption. This kind of bugs are called no-sleep bugs [44]. Moreover, the misused of these API can lead to energy bugs that cause a great amount of power consumption.

2.1.5 Control Flow Graph

In this thesis, Control Flow Graphs (CFG) will be used to perform different kind of analysis. A CFG is a directed graph the represents all the possible execution paths that an application can have. In this graph nodes are basic blocks and edges represents control flow from one basic block to another [14]. A CFG is build for a particular method (intra-procedural), wheres an Inter-procedural Control Flow Graph (ICFG) takes in account methods calls and transfers the control to other methods (inter-procedural).

2.2 Related Work

2.2.1 Security Analysis using Program Analysis for Android

Security is another important problem that smartphones users face. Android, giving the openness of its market, is one of the most affected mobile

operating systems by security vulnerabilities. That is why, security is one of the areas that has more research work. Researchers apply techniques such as static analysis, dynamic analysis and testing to discover different kind of security vulnerabilities in Android applications.

Permissions

Felt, et al. [24] build the tool Stowaway to detect if an Android application is overprivileged (declares more permissions in the *AndroidManifest.xml* file than it needs). They first map all API calls to permissions required by these calls and then use this map to detect all the permissions that the Android application needs. These permissions are checked against the ones declared by the developers in the *AndroidManifest.xml*.

Security Vulnerabilities

Several efforts has been done to identify vulnerabilities regarding privilege escalation attacks including [19, 20] and also proposing new techniques [47]. Moreover, privacy leaking is another problem that has been thoroughly studied by [21, 29, 33, 37, 60, 61]. Other approaches such as model checking (SMT solvers) have been used to identify logical security errors in Android applications as well [38].

Malware Detection

Most of the previous approaches try to identify security vulnerabilities on Android applications. However, *RiskRanker* [30] and *DroidMat* [58] try to identify malicious behaviors directly on applications from the market using static analysis. Moreover, Alazab et al. [13] develop *Droidbox* to classify be-

nign and malicious applications using dynamic analysis. Different techniques are also applied for malware detection. For instance, Shabtai et al. [49] use machine learning to identify malicious behavior patterns.

Interaction between Components

There are different research problems that arise from the interactions between components. For example, Chin et al. [23] examine the interaction of components in Android applications in order to identify application communication vulnerabilities. Their work tries to find whether an Android application is vulnerable to attacks based on inter-component method calls from malicious applications. In addition, Chaudhuri [22] and Fuchs et al. [26] create a typed language to model interactions between Android main components to reason about the dataflow security properties of the application.

2.2.2 Energy Related Analysis

As it was mentioned before, power consumption is one of the main problems faced by smartphones. There are different approaches to identify bottlenecks and bugs related to power consumption. One approach includes profiling and debugging techniques to discover where Android applications spend more energy. For instance, *Eprof* [44] is a fine grained energy profiler which is capable of finding energy bugs and also capable of point these bugs in the source code of Android applications. In addition, *eCalc* [31] and *eLens* [32] uses cost functions–they compute the estimated energy cost for each function based on the type of instruction–to estimate the power consumption of an

Android application using execution traces generated from the application.

Another approach is to apply static analysis on Android applications to check energy policies or protocols from the Android API. In this line, Pathak, et. al. [42] adapt a reaching definitions dataflow analysis to detect no-sleep bugs checking if all the acquired *wake locks* are released through all the possible paths. Vekris et al. [56] also uses a dataflow analysis to find bugs related to wake locks using policies based on defined exit points in components lifecycles where *wake locks* must be released. Both approaches try to model the order of callback execution using the components lifecycles but they do not take in account interactions between components.

2.2.3 Specification Mining

Mining specifications for program verification has become an important technique to automatically learn API protocols. During the last years a lot of research has been made in this field [15, 27, 28, 46, 50, 57, 59].

There are two main directions on mining specifications: dynamic analysis and static analysis. Most of the research done in this area uses a dynamic analysis approach. However, in order to use dynamic analysis, the application must be instrumented and then run in order to get dynamic traces from it. This task can be executed manually or automatically (increase considerably the complexity of the approach).

On the other hand, specification mining using static analysis does not need to run the program, but it suffers of problems such as aliasing, infea-

sible paths and path exploitation (for big applications in lines of code and complexity). It can be classified as component-side and client-side [50]. A component-side approach uses directly the source code of the API, whereas a client-side approach utilizes client applications of the API.

In this thesis, a client-side static analysis specification mining approach will be used. The clients will be Android applications obtained from the Google Play market.

2.2.4 Modeling Event-driven Systems

It was shown that Android is a very dynamic system in which different components can be taken as entry points and inter-component method calls can impact the control flow of and Android application. Therefore, in order to statically analyze Android applications, we have to model this dynamic behavior to recreate the control flow.

Research has been done on other kind of event-driven systems. For instance, Nguyen et al. [41] and Cheung et al. [34] focus on modeling Wireless Sensor Network (WSN) applications. WSN are event-based, concurrent systems that like Android are not easy to model. Both papers [41] and [34] try to define an accurate representation of WSNs.

Although, Android and WSNs applications are concurrent and event-based, they have different behaviors. Therefore, the approach to model Android applications will not be based on the models developed for WSNs.

Chapter 3

Mining Energy Specifications (E-Specs) and Energy Bugs (E-Bugs)

The first approach that is going to be presented in the thesis, is a intra-component control flow model. In this approach, just the components lifecycles are taken in account, therefore, the interaction between components does not have an impact in the control flow of the representation. Moreover, a static analysis framework to mine energy related specifications in Android applications is developed. We will use a summary approach applying analysis to each callback by separated and then composing the results based on the components lifecycles. The framework is composed by three main phases: the generation of an intermediate representation of the Android application, the specification miner and a post-processing phase (see figure 3.1).

3.1 Modeling Intra-Component Control Flow

To model intra-component control flow of Android applications we are going to use the lifecycles describe in section 2.1 (figure 2.2 shows the lifecycle for activities and figures 2.3a and 2.3b show the lifecycles for services). These lifecycles give us a control flow of callbacks for each component, therefore, we

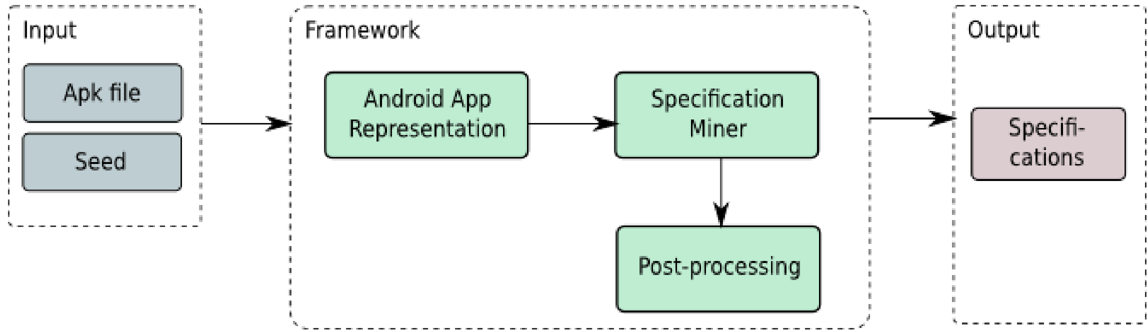


Figure 3.1: Framework

can simulate their behavior. For each callback we are going to generate an ICFG. Theses ICFGs will be the inputs of the specification miner.

3.2 Specification Miner

An inter-procedural, path-sensitive, context-sensitive, value-flow analysis is implemented in order to mine specifications. This analysis will take an ICFG of a callback and a seed (which is a class from the Android API) and generates a set of sequences of methods calls for each different instance of the seed class found in the application.

The analysis must be path-sensitive since all the possible execution paths of the application must be explored. By looking at all the execution paths, we can discover whether there are paths where the specification mined is broken. This approach has an advantage over testing since it covers all the code and in testing, there might not be enough test cases to cover all the code. However, this approach suffers from path exploitation when the application is

big in terms of lines of code and complex in terms of possible execution paths. Therefore, we must apply a mechanism to reduce program space but without losing precision (see section 3.4).

Another aspect of the analysis is that when recollecting the methods of the seed API, the analysis must recognize to what particular object instance the invocation belongs. Therefore, we must keep track of each instance of the seed class through the program taking in account all the aliases the reference can has. To solve this problem we have developed a value flow analysis that tracks the different instances of the given seed class through each path in the application. This analysis finds the creation of objects of the seed class and tracks how its value is assigned to different variables through the ICFG. Therefore, the analysis keeps all the aliases of the instance and use this information to determine the instance of a particular method call of the seed class.

The result of this analysis must be a set of sequences in which each sequence represents a list of methods calls of the seed class for a particular path. Since the input is just a callback of a component, this sequences must be composed with other sequences. The algorithm developed takes as inputs the sets of sequences for each callback and the lifecycles of activities, services and broadcast receivers. Then, it composes the sequences of callbacks of the same component according to the lifecycle of the component using the information gathered by the value flow analysis to avoid composing sequences of different instances.

3.3 Post-Processing

After the specification miner finishes, we have specifications for each of the applications analyzed. We analyzed different versions of the same application because we wanted to compare if the specifications were consistent across all the versions. All the specifications found for one version were compared against each other version of the same application. In addition, we take all set of sequences mined for all the versions and build a prefix tree acceptor (PTA). A PTA is an automaton similar to a tree where every leaf is a accept state and it is used by algorithms of specification learning such as k-tails FSA [36]. As future work we can apply different algorithms, such as k-tails FSA, to generate a final specification for energy related APIs.

3.4 Implementation

3.4.1 Getting Intermediate Representation of Android Applications

The framework described before, takes an intermediate representation (ICFG) of the application it analyzes. Android applications are packaged in Android application packages (APK) files which contains the bytecode of the application (bytecode for the Dalvik VM) and its assets. Since most of the tools to work on Java applications uses either source code or Java bytecode for the Oracle JVM, Dex2Jar [7] is used to transform an APK file to a Jar file. However, we still need an intermediate representation for our Specification Miner. In order to get an intermediate representation from the Jar file we use Soot.

Soot is a framework that was created for optimizing Java bytecode [54]. It offers four intermediate representations from which we chose Jimple [55]. Jimple is a 3-address code (TAC), stackless intermediate representation. In addition, since Soot offers also a CallGraph for the application, we can build an ICFG for each callback of each component which are used by the specification miner.

3.4.2 Optimizations

Path-sensitive, context-sensitive is a very expensive analysis. We apply some optimizations to make the analysis scalable. For instance, to reduce the space of the analysis, we apply a linear marker, which is a path-insensitive analysis that scan through each statement in the ICFG of the callbacks and marks the methods that contains method calls of the class under analysis (seed) and the methods that call these methods. Figure 3.2 shows a CFG of a simple activity (figure 3.3 shows its source code) for an Android application. In this control flow graph, we see the implementation of the callback *onResume* which creates an object of the class Example. The Example class has the methods **a**, **b**, **c**, **d** and **e** where just **a** and **e** have calls of the class *PowerManager.WakeLock* (the input seed). In this case, the linear marker goes through each statement marking all the method that has calls of the seed, the method *obj.a()*. Since, the linear marker works inter-procedural, the method *obj.d()* is marked as well because it has a call to the method *e()* which contains calls of the seed class (see figure 3.3 line 34).

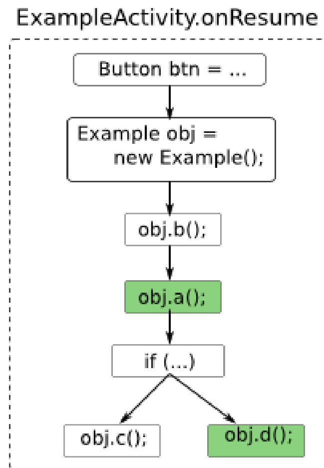


Figure 3.2: CFG example for linear marker

3.4.3 Seed Classes from the Android API

The analysis was performed using six different classes from the Android API as seeds. We focus on classes that perform any kind of lock on components of the device such as CPU, GPS, Wifi radio, sensors, keyword and the camera. The seeds we use are the following:

- *android.os.PowerManager.WakeLock*: the application can acquire a wake lock on the CPU and even on the screen of the device.

- *android.net.wifi.WifiManager.WifiLock*: with this lock, the application can apply a wake lock to the Wifi radio. This is useful for streaming applications when the device has access to Wifi networks since the power consumption is reduced significantly (tail energy is comparable, but the data transfer is faster [17]).

```
1  public class ExampleActivity {
2    protected void onResume() {
3      Button btn = ...;
4      Example obj = new Example();
5      obj.b();
6      obj.a();
7      if (...) {
8        obj.c();
9      } else {
10       obj.e();
11     }
12     ...
13   }
14 }
15
16 public class Example {
17   PowerManager.WakeLock lPartial;
18   PowerManager.WakeLock lScreen;
19   public void a() {
20     lPartial = PowerManager.
21                  newWakeLock(...);
22     lScreen = PowerManager.
23                  newWakeLock(...);
24     lPartial.acquire();
25     lScreen.acquire();
26   }
27   public void b() {
28     ...
29   }
30   public void c() {
31     ...
32   }
33   public void e() {
34       d();
35   }
36   public void d() {
37     if (lPartial.isHeld())
38       lPartial.release();
39   }
40 }
```

Figure 3.3: Source Code example for linear marker

- *android.location.LocationManager*: use to request updates of location using the GPS or other techniques. If the request of updates is not removed, then it can keep asking for location updates even when the application is not been used.

- *android.hardware.Camera*: used to perform different actions with the camera. The application can also lock the camera to exclusive use and has to unlock it when it finishes the action it was doing.

- *android.app.KeyguardManager.KeyguardLock*: blocks the keyboard during the execution of the application. The application has to release the lock once it has finished the task it wanted to do without the keyboard.

- *android.hardware.SensorManager*: use to access the different sensors the device has. After the application finishes using a sensor, it must disable it [48].

3.5 Experimental Results

The specification miner analysis was applied to 13 applications from the Google market and a number of different versions of each application. Our experiments were run on a VM with AMD Opteron 6204 3.30 GHz (two CPUs) and 64GB of RAM.

Table 3.1 shows the applications we analyze; the number of versions; the total number of classes, methods and fields (static and instance fields) per application including all the versions–it is worth to mention that these are

approximate values since we are getting these numbers from a Jar file that was reversed engineered from an obfuscated Dalvik VM bytecode.

Table 3.1: Benchmarks

Applications	**No. of Versions**	**No. of Classes**	**No. of Methods**	**No. of Fields**
Netflix	9	2322	15672	10555
DroidNotify	13	4331	32907	23194
MyTracks	3	1655	11726	6108
PowerManager	3	538	2895	3740
Skype	6	7859	32778	22594
Noled	4	751	3262	3703
WidgetLocker	3	2057	10046	7288
NYTimes	2	826	5854	3021
Missed Call Pro	2	836	6991	1951
Foursquare	3	3273	22059	17671
BLN	2	269	1371	799
UCam Ultra Camera	4	4449	30743	17188
Facebook	8	17793	107099	65132

The specification miner and post-processing phases were applied on the applications listed before getting acceptable results in terms of scalability. Table 3.2 shows the results of the specification miner analysis. The column *Common FSM* shows the number of finite state machines found that were present across all the versions. The *Total Time* column shows accumulate time in seconds that the specification miner took for all the versions and all the seeds for each application.

For the majority of the applications, the analysis took less than 3 hours to run through all the seeds and all the versions. Figure 3.4 shows the relation between the number of classes per application and the time the analysis took. With the exception of *Noled*, the time spent by the analysis increases according to the number of classes the application has–there are different factors

Table 3.2: General Specification Mining Results

Benchmark	No. of Versions	Unique Specs	Common Specs	Total Time (seconds)
Netflix	9	4	2	3806
DroidNotify	13	10	2	9392
MyTracks	3	4	4	2211
PowerManager	3	2	0	1605
Skype	6	1	0	4994
Noled	4	5	5	18802
WidgetLocker	3	14	7	1842
NYTimes	2	2	2	916
Missed Call Pro	2	1	1	1244
Foursquare	3	5	3	6086
BLN	2	5	3	606
UCam Ultra Camera	4	23	17	7572
Facebook	8	20	2	14223

that may have contributed to the performance results on *Noled* such as the complexity of the application in terms of possible paths.

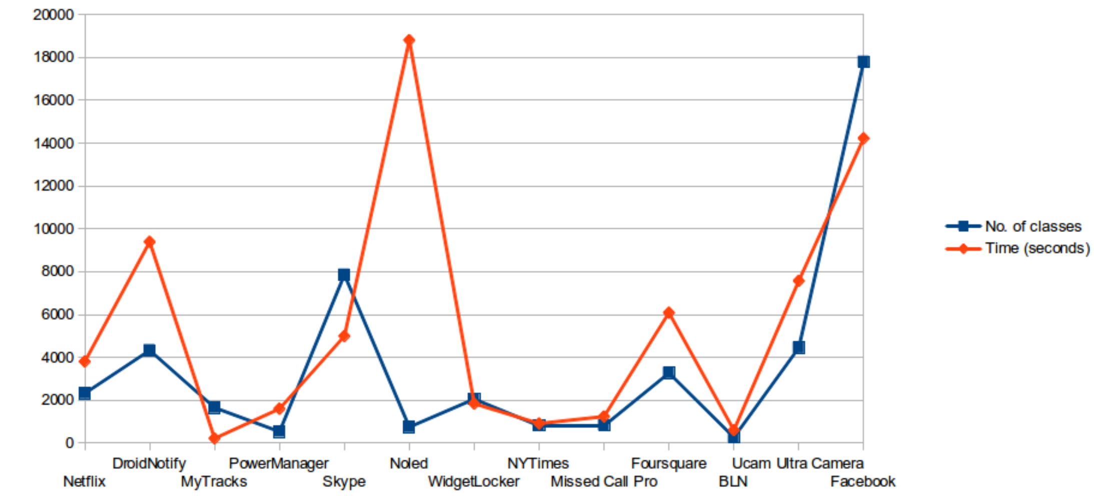


Figure 3.4: Performance: No. of classes vs. Analysis time

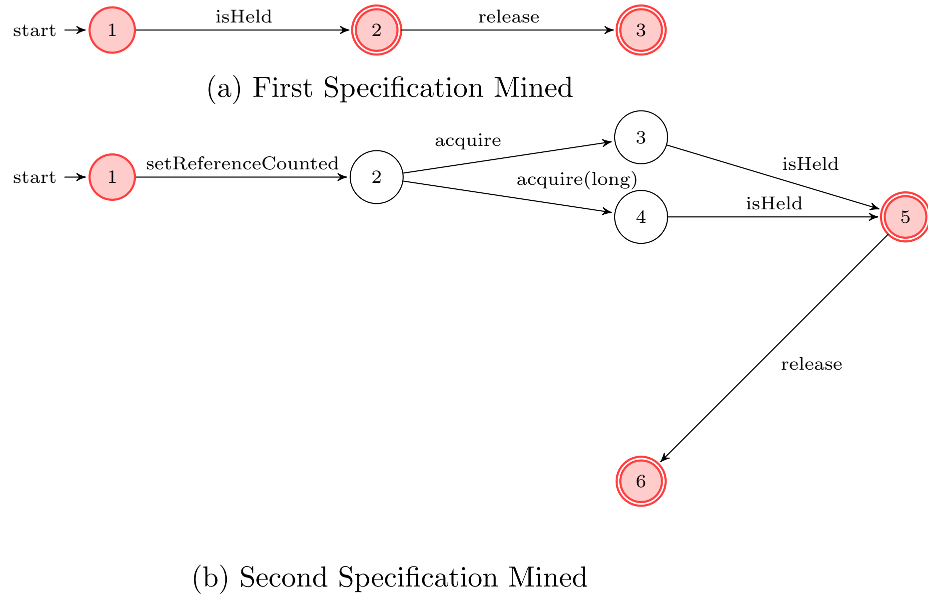
In addition, we found that across multiple versions of the same applica-

tion and multiples components of one application, the number of specifications found for the given seeds does not vary often (they are simple calls to acquire and release locks on components of the device). The only exception found was the specifications for the class *android.hardware.Camera* which can be used for different tasks.

```
1  class UIWebViewActivity extends NetflixActivity {
2    public void onStart() {
3      stayAwake();
4      ...
5    }
6    public void onStop() {
7      releaseAwakeLock();
8      ...
9    }
10 }
11 class NetflixActivity extends Activity {
12   public void stayAwake() {
13     if (this.wakeLock != null) {
14       if (this.wakeLock.isHeld()) {
15           this.wakeLock.release() {
16       }
17       this.wakeLock = null;
18     }
19     this.wakeLock = p.newWakeLock(getWakeLockFlag(), getLockName());
20     this.wakeLock.setReferenceCounted(false);
21     if (getLockTimeout() > 0) {
22       this.wakeLock.acquire(60000L);
23     }
24     else {
25       this.wakeLock.acquire();
26     }
27   }
28   public void releaseAwakeLock() {
29     if (this.wakeLock.isHeld())
30       this.wakeLock.release();
```

Figure 3.5: Source Code

The main focus of the evaluation of this project was to find whether the



(a) First Specification Mined

(b) Second Specification Mined

Figure 3.6: Example of results

intra-component approach for modeling Android applications was suitable to perform static analysis and specially specification mining. We found that for some applications given the logic the programmers used, the intra-component approach was good enough. For instance, figure 3.5 shows the part of the found in all the versions of the *Netflix* Android application use for the experiments. In this case, our analysis works perfectly mining two different specifications from the code across the different callbacks of an activity. Figure 3.6a is mined from the lines 14-17 in 3.5. Since, in the line 19 a new instance of the class *android.os.PowerManager.WakeLock* is assign to the variable *wakeLock*, the analysis recognizes all the following calls as part of a different instance of the previous method calls. That is why the specification 3.6b is generated.

In addition, table 3.3 shows the number of false positives found per

Table 3.3: False Positives for the Specifications

Application	False Positives	Total Specs	Percentage False Positives
Netflix	1	4	25.00%
DroidNotify	5	10	50.00%
MyTracks	0	4	0%
PowerManager	2	2	100.00%
Skype	1	1	100.00%
WidgetLocker	7	14	50.00%
NYTimes	2	2	100.00%
MissCallPro	1	1	100.00%
Foursquare	3	5	60.00%
BLN	2	5	40.00%
Ucam Ultra Camera	13	23	57.00%
Facebook	13	20	65.00%

application. We discover that most of the causes of false positives and false negatives were related to threads, inn-accurate given by the value flow (aliasing), instances of the seed class under analysis that were used in different components (inter-component specifications), and callbacks of GUI and sensor event callbacks out of the lifecycles used for each component.

Inter-component specifications

We found that the intra-component approach did not work for some applications. There are some applications in which instances of the seed class flow through different components. In other cases, the applications use a helper class with static fields (including static fields for the wake lock objects) and static methods that are called through different components. In addition, there are other callbacks that must be taken in account besides lifecycle callbacks. For instance, GUI and sensors callbacks are important to take in account in order to improve the code coverage. Therefore, inter-component specifications

and GUI and sensors callbacks must be analyzed in order to improve the accuracy of the results.

Aliasing

Aliasing is one of the main problem during mining specifications on object oriented languages. Objects under analysis can flow through complex data structures and keep track of them is a complex task. Therefore, for analysis of object oriented languages having a precise information about how the objects flow is important for the accuracy of the results.

Threads

Android applications are concurrent in nature. In Android, every application runs on its own main thread (called UI thread). If the application tries to do a task that runs for a considerable amount of time (normally more than 2 seconds), the system will raise a dialog asking the user if he/she wants to close it or wait–this is called Application Not Responding (ANR) mechanism. That is why applications normally assign long-time running tasks to other threads creating false positives and false negatives in our results. Thread analysis is not the main purpose of this work, therefore, it can be considered as future work.

3.6 Conclusion

In conclusion, in this chapter an intra-component specification mining has been developed to analyzed Android applications. We discovered that for some applications, the intra-component approach was good enough, whereas,

for other applications, an inter-component approach is needed to mine specifications. In addition, we discovered some sources of imprecision in the analysis that can be tackle as future work.

Chapter 4

Building a Representation for Analyzing Android Applications

4.1 Motivation

As it was mentioned in the previous chapter, the main causes of false positives and false negatives we discovered during analyzing Android applications were threads, inaccuracies of value flow or/and points to analysis, and inter-components specifications (specifications that are found in more than one component). From the three causes, thread analysis [51], aliasing [16, 18, 35, 40, 53] are general static analysis challenges. However, we have not found a work that tries to model Android applications taking in account inter-component control flow. Therefore, in this chapter we define an inter-component approach to analyze Android applications

As a motivating example, figure 4.1 shows a source code found in *Droid-Nofity* in which a powermanager wake lock object is instantiated and **acquired** in a broadcast receiver. Then, the broadcast receiver starts a service which performs some job and **releases** the same wake lock acquired previously. In this case, the intra-component approach used to model Android applications and the specification miner will mine two different specifications, one in the

broadcast receiver and the other in the service– both mined specifications are incomplete and therefore are false positives.

```
1  class CalendarNotificationAlarmReceiver ... {
2   public void onReceive(...) {
3       Intent localIntent = new Intent(paramContext,
4                                  Service1.class);
5       localIntent.putExtras(paramIntent.getExtras());
6       WakefulIntentService.sendWakefulWork(
7          paramContext, localIntent);
8       ...
9    }
10 }
11 class WakefulIntentService extends IntentService {
12  public static void sendWakefulWork(Context context,
13        Intent intent) {
14     Common.acquirePartialWakeLock(context);
15     paramContext.startService(intent);
16  }
17  public onHandleIntent(...) {
18     doWakefulWork(paramIntent);
19     if (!Common.isFullWakelockInUse())
20       Common.clearWakeLock();
21  }
22 }
23 class Service1 extends WakefulIntentService {
24     ...
25 }
```

Figure 4.1: Inter-component specification

4.2 Modeling Inter-Component Control Flow

In this chapter, a model that takes into account both, components lifecycles and interaction between components is going to be developed. We should model the behavior of Android applications based on three of its main components–Activity, Service, BroadcastReceiver–and its interactions. Because Android primarily is an event-driven system, we model the system using

event-driven finite state machines since they help us to represent the knowledge we have from the system–the knowledge is represented by signals or inputs which trigger actions that change the state of the system [25]. We have studied the Android system and found signals related with lifecycles and interactions of the three components mentioned before. We will represent events related with the system's environment such as graphical user interface events (GUI), sensor events and others as external signals. In addition, exceptional events, such as system out of memory, are categorize as external signals. Internal signals are represented by inter-component method calls which basically start an interaction between two components in an Android application: the caller and the callee. For instance, *startActivity* launches an activity to the foreground. Another example is how a component can use the method call *sendBroadcast* to interact with broadcast receivers. Moreover, in this document we state certain constraints that are applied to these calls.

4.3 Modeling Android using Event-driven Finite State Machines

We model the Android system (we are going to use Android from now on to refer the Android system) as a control system that uses input events (signals) to determine the behavior of the application. We use event-driven finite state machines to model the behavior of three of the four main component in Android. The signals are organized in two categories: internal and external signals.

4.3.1 External Signals

We define external signals as events generated by the environment of an Android application and not an event made directly by the application. For instance, when the user tabs the *home button* in an Android phone, the system stops the current application (activity) and triggers the home application–tabs to the back button is also taken as external signal. Another external signal is when the system ran out of memory and release some memory sometimes releasing activities and services that are kept in memory affecting the lifecycle of these components. In addition, all events related with sensors and GUI are categorized as external. All these external signals have a direct impact in the lifecycles of components.

4.3.2 Internal Signals

In contrast to external signals, internal signals are generated by method calls found in the application's code. We consider, as internal signals, all the methods calls that affects the behavior of components in Android. We categorize inter-component method calls (methods that allow the interaction between two components) as internal signals since they also affect the behavior of components. For instance, when an activity wants to do a long-running task in the background, the activity can call a service to do that job. The activity can uses inter-component calls *binService* or *startService* which triggers actions to create and run the service. The following list, show the inter-component method calls we analyzed to build our model:

- for activities, the methods *startActivity*, *startActivityIfNeeded* and *startActivityForResult* can be used to call activities.

- for services, Android defines the methods *startService*, *stopService*, *bindService*, *unbindService.*

- to call broadcast receivers, the Android API has *sendBroadcast* and *sendOrderedBroacast* (there are other variants of these two methods that does not have an effect in our models).

In the following sections, we will define what is the behavior of each method call and what constraints these calls have. In addition to inter-component method calls, we also consider method calls that belong to the component class which affect its lifecycle. For instance, in activities, when the method *finish* is called, the system stops the activity immediately. Moreover, for services the method *stopSelf* stops the service after it is called.

4.3.3 How External and Internal Signals are treated in our model

First of all, our main goal is to build an intermediate representation of Android applications. Therefore, our model will take in account all the possible execution paths the applications can have. In that regard, we will assume that external signals will be send, contrary to internal signals that must be explicit in the application's code.

4.4 Models for Components

To model the behavior the three components mentioned and their interactions, we have to identify which external and internal signals affect their behavior. Following, models for activities, services and broadcast receivers are presented.

4.4.1 Modeling Activities

Activities are the main components in Android applications. They normally provide an user interface and are the main entry points for an application. Activities can be triggered by internal and external signals. We also have to take in account that these signals has different behaviors depending on the state of the activity.

Activities can be activated and re-activated. For instance, when an user tabs on the launcher for an application and the application has not been run before (it is not in memory), the system will look for the main activity declared in the *AndroidManifest.xml* file and launch it. If the application is going to be accessed again (now the application is in memory), the system will restart that last activity used. The following models show these and more cases for activities (section 2.1).

Launch Activity. As we said before, the main activity declared in the *AndroidManifest.xml* file is launched when the user tabs the launcher icon of the application. The other activities can be launched using the inter-component

call *startActivity*, *startActivityIfNeeded* or *startActivityForResult*. Figure 4.2 shows the behavior of an activity when it is launch as main activity or using *startActivity* from other components different of activities–services and broadcast receivers. Once the activity is *Active*, it can receive events related to GUI, sensors. It can also be relaunched (see General Model 4.4.1.1 for more information).

Constraint: when the *startActivity* method is called from another component, which is not an activity, the intent must have the flag *FLAG_ACTIVITY_NEW_TASK* [1].

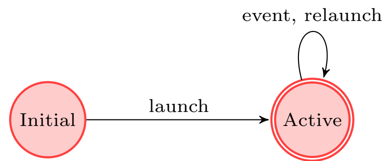


Figure 4.2: Activity Launch Task State Machine

When an activity is launched from another activity, the system will pause the caller activity, then launch the callee activity and ultimately stop the caller activity (see figure 4.3).

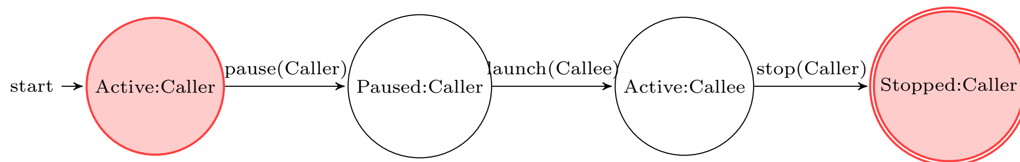


Figure 4.3: startActivity from Activity

Pause and Stop Activity. When an activity is overshadowed by another

activity, but it is still visible, the system just pauses it. Once, the second activity is closed, the system executes the action *resume* to active the previous activity. If the user is leaving the application (because of tab on the home button for example), the system will execute the actions *pause* and *stop* in sequence. In addition, the *finish* method in activities stop them. Therefore, if this method is called when the activity is active (or in process to be active), the system will execute the *stop* action immediately.

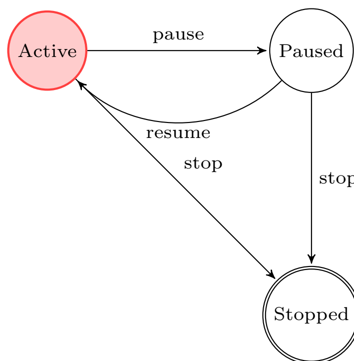


Figure 4.4: Pause and Stop Activity

Restart Activity. Once an activity is in memory (*stopped*), every time it is called again, the system will execute the *restart* action. This action is similar to *launch*, but it will not execute the callback *onCreate*. Instead it executes the callback *onRestart* (see section 4.4.1.1).

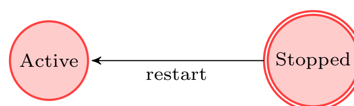


Figure 4.5: Restart Activity

Destroy Activity. If the system runs out of memory, it can release some

components from memory including activities. An activity can be released just if it is in the *stopped* state [1]. When the activity is released from memory, the system executes the action *destroy*, which executes the callback *onDestroy* in the activity.

Figure 4.6: Destroy Activity

4.4.1.1 General Model

Figure 4.7 shows a general model of the behavior of activities. This is a composed model of all the previous models presented. Following, each of the states and actions in the finite state machine are described.

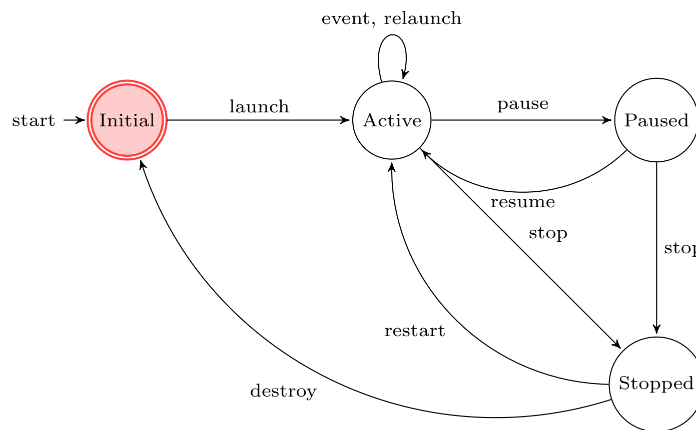


Figure 4.7: General Model for Activity

Description of states

- **Initial**: the activity has not been called or it was released from memory.
- **Active**: the activity is visible and allows interaction with an user.
- **Paused**: the activity is paused by different reasons. One is as part of its life-cycle. When the activity is going to be stopped, the system pauses the activity first and then stop it. The other reason is when another activity has opaque the current activity, although it is still visible [1].
- **Stopped**: the activity is not visible. There is another activity that is completely in the foreground. However, the system keeps the activity in memory.

Description of actions

- **launch**: this action can be triggered by different signals. For instance, if the activity is declared in the *AndroidManifest.xml* file as the main activity, this activity will be launch at the time the user access the application (external signal). The other signal is when the activity is started using the inter-component method call *startActivity* (internal signal). The launch action has the following sequence of callbacks: $onCreate \rightarrow onStart \rightarrow onResume$.
- **event**: once the activity is active, it can handle different events such as graphical user interface (GUI) events, sensor events and others. This action will have just one method associate and it will be the handler associate with the event (e.g. *onClick* for buttons).

- **relaunch**: the action is normally triggered when the user change the orientation of the phone. When the orientation changes, the system will pause, stop, destroy the activity and then launch it again. Therefore the sequence of callbacks will be the following: $onPause \rightarrow onStop \rightarrow onDestroy \rightarrow onCreate \rightarrow onStart \rightarrow onResume$ (there are other callbacks executed which we do not take in account for our model).

- **pause**: normally when the activity is going to be stopped, the system will trigger the action *pause* first. This events can be caused because of different signals such as the user clicks that home button or back button. In addition, if there is a *startActivity* call, then the system pauses the activity before the new activity is launched (see figure 4.3). The *pause* action just has one callback associated, $onPause$.

- **stop**: there are two states where the action *stop* can be triggered from. If the method *finish* is called within the activity, then it will close the activity and triggers the stop action. In addition, if the method is called in any of the launch callbacks, it will stop the execution of the following callbacks in the action and it will trigger the action *stop* immediately. The other state the action stop can be triggered from *Paused*. This will happen as part of the activity's lifecycle when an user click the back or the home button and the system puts in the background the current activity (there are another scenarios). It is worth to mention that if the *pause* action is triggered because of a *startActivity* signal, then the stop

action will not be triggered until the started activity has been launched. The callback associate to the *stop* action is *onStop*.

- **resume**: if the activity loses focus to another activity but it is still visible, the system will pause the activity. Once it recovers the focus, the system will trigger the action *resume* to active the activity again. This action executes the callback *onResume*.

- **restart**: when the activity is stopped but is still in memory, the system does not have to create it again. Therefore, when the activity is accessed again, the system restart the activity instead of launch it again. The action execute the following sequence of callbacks: $onRestart \rightarrow onStart \rightarrow onResume$.

- **destroy**: this action will be triggered when the system destroys the activity held to release memory. The activity can be killed just when the activity is in the state *Stopped* [1].

4.4.2 Modeling Services

In contrast to activities, which normally offer a user interface to interact with the user, a service is used for long-running tasks that are executed in the background [4]. Services can be run just using inter-component calls (internal signals). Following, all the inter-component calls related with services are explained.

startService. Figure 4.8 shows the model for calling *startService* when the service (callee) is not created (running). After the service is created, all the *startService* calls will just trigger the action *start*.

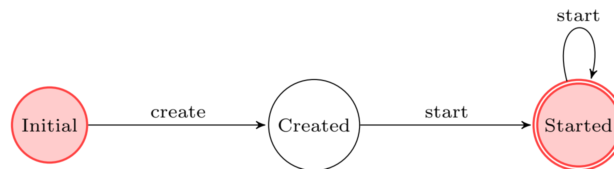


Figure 4.8: startService when service is not running

In case the service has been bound, a call to the method *startService* will change the state of the service to *started and bound*. Figure 4.9 shows this behavior.

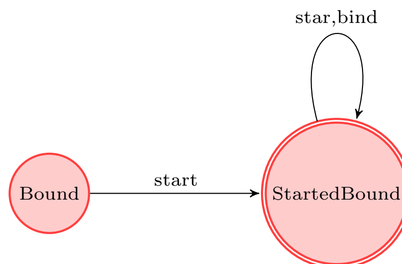


Figure 4.9: startService when service is bound

stopService. In case there is a *stopService* inter-component call, the callee service will be stopped–this call just triggers the action *stop* (see model in figure 4.14).

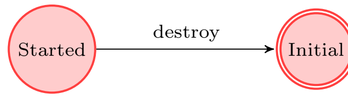


Figure 4.10: stopService when service is Started

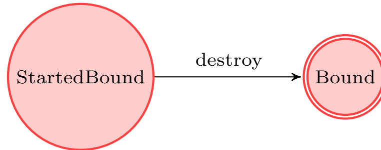


Figure 4.11: stopService when service is Started and Bound

Constraints: the service must be *started* or *started and bound*.

bindService. Figure 4.12 shows the model of the behavior when the inter-component method call *bindService* is executed and the service has not been created. Once the service is *bound*, all the calls *bindService* will not affect the state of the component.

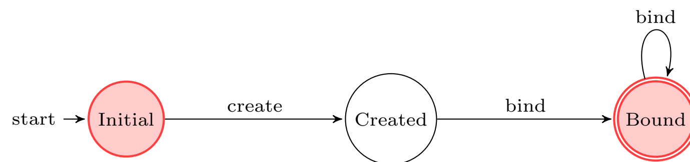


Figure 4.12: bindService when service is not running

If the service has been started using the inter-component call *start-Service* and the call *bindService* is executed, the system will trigger the *bind* action and the service will pass to the state *started and bound* (see figure 4.13).

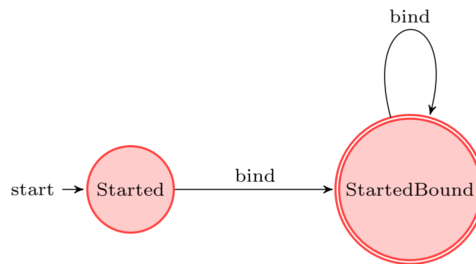


Figure 4.13: bindService when service is started

Constraints: broadcast receivers cannot bind a service because the life span of a broadcast receiver is too short. It is recommendable that if a broadcast receiver wants to work with services, it must call *startService*. Another constraints is that there must be an *unbindService* call for each component that bind the service. For example, if an activity binds a service, and then it is stopped without unbinding the service, the service will remain running but without interacting with the activity– if the programmer wants that behavior, it is better to use *startService*.

unbindService In case there is a *unbindService* inter-component call, the callee service will be unbound–this call just triggers the action *unbind* (see model in figure 4.14). If there is no other component bound to the service, the system will trigger the action *destroy*.

Constraints: the service must be *bound* or *started and bound*.

4.4.2.1 General Model

The general model is the composition of all the model described in the previous section. Following, the description of each state and action in this model.

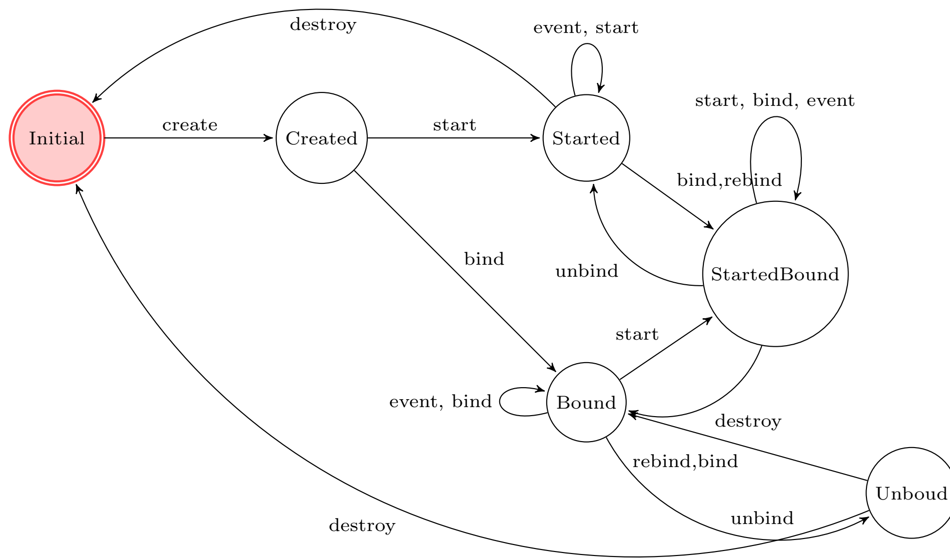


Figure 4.14: General Model for Service

Description of states

- **Initial**: the service has not been called or has been released from memory.
- **Created**: the service was created and it is in memory.
- **Started**: the service is running in the background until the inter-component

call *stopService* is executed or the method *stopSelf* is called. The service is in this state after a *startService* inter-component call has been made.

- **Bound**: the service is bound to another component through the inter-component call *bindService*. The service will be running until the component that bound it call the inter-component call *unbindService*. It is worth to mention that multiple components can bind a service at the same time. Therefore, the service will run until the last component that bound it unbind it.

- **Started and Bound**: the service has either started first and then bound or bound first and then started. In this case the service will be maintain alive by two ways. Until the actions stop and unbind (in either order see figure 4.14) are triggered, the service will not be destroyed.

- **Unbound**: the service is unbound from the component that bound it. Since the service can be still running, the same component can bind it again– the rebind action can also be triggered in case the *onUnbind* callback returns true.

Description of actions

- **create**: when a service is called, either by the inter-component calls *startService* or *bindService*, if the service has not been called before, the system will create the service first. This action will execute the callback *onCreate* on the service.

- **start**: this action is triggered by the inter-component call *startService*. The action will execute the callback *onStarCommand*. If the service is an *IntentService*, then the *onHandleIntent* callback will be executed after *onStartCommand*.

- **event**: this action is triggered by any event that is handle in the service. For instance, a service can register handlers for network notifications.

- **bind**: inter-component calls of *bindService* will trigger this action. As it was mentioned before, the *bindService* call first check if the system was created, and then triggers this action. This action executes the callback *onBind*.

- **unbind**: this action is triggered by the inter-component call *unbindService*. This action executes the callback *onUnbind*.

- **rebind**: if the *onUnbind* callback returns true, the next time a component tries to bind the same service again, the system will trigger the action *rebind* instead of *bind*.

- **destroy**: this action can be triggered by a call of the method *stopSelf* or an inter-component call *stopService*. This action executes the callback *onDestroy*. In addition, after the *unbind* action if there is no other component bound to the service, the system will execute the action *destroy*. The are external signals that can trigger the action *destroy*. For instance, when the system runs out of memory, it can destroy services that are

running. In addition, users can stop services from the configuration of the phone. Therefore, we will assume this action, even if there is not an *stopService* or *stopSelf* call.

4.4.3 Modeling Broadcast Receivers

Broadcast receivers just have one action, *receive* The action *receive* executes the callback *onReceive*. Once the *onReceive* callback finishes, the broadcast receiver is no longer active. A component can communicate with broadcast receivers by sending messages using the methods *sendBroadcast* and *sendOrderedBroadcast* (there are other verions of these methods which make no difference in our model). These two methods can send broadcast messages to more than one broadcast receiver because the intent defined can match zero o more components. However, they have different behaviors in the order of execution.

Broadcast receivers can be also called by the system. When a broadcast receiver is declared with the action *android.intent.action.BOOT_COMPLETED* in the *AndroidManifest.xml* file, the action *receive* will be triggered when the system boots. Figure 4.15 shows the model for broadcast receivers.

Figure 4.15: BroadcastReceiver Task State Machine

As we said, *sendBroadcast* and *sendOrderedBroadcast* are used to communicate with broadcast receivers. Here we explain their behavior:

sendBroadcast. This inter-component call will execute the action *receive* for each component that match the intent. The order of execution of the broadcast receivers will be random.

sendOrderedBroadcast. In case the call is *sendOrderedBroadcast*, the system will execute the *receive* action for each component in order. The order is ascending based on the property *priority* defined in the *AndroidManifest.xml* file for each broadcast receiver. In case two or more broadcast receivers have the same priority, the order for those components will be random.

4.5 Representation

In the previous section, the behavior of Android applications, and specifically of activities, services and broadcast receivers, was describe using finite state machines. The main goal of this work is to generate an intermediate representation that takes component's lifecycle and inter-component control flow in account. In this section, an intermediate representation for Android applications is presented. This representation is built using the previous models and the source code of the application. Definition 1 describes an Android Inter-component Graph (AIG).

Definition 1. *An Android Inter-component graph (AIG) for an application a, is a directed graph* $G =< N; E >$ *where* N *is a set of nodes that represents callback components and* E *is the set of edges that represent the control flow between callback components.*

4.6 Construction of the AIG

The following steps define how to build the AIG for an application:

1. Find the main activity in the *AndroidManifest.xml* file.
2. Build the *startActivity* model using the main activity (section 4.4.1).
    (a) At the end of each transition in the model, all the internal signals found in the action (inter-component method calls found in the callbacks that belong to the action) associated with the transition are processed. This process consist on finding what component(s) the inter-component method call targets and the model for the inter-component call. Then, this model is built recursively using the target component(s). It is worth to mention that the constraints mention in the previous section (4.4. are checked for any violation.
    (b) Lastly, change the state of the target component(s) based on its current state (the analysis keeps the state of each component) and the inter-component call. For example, if a service A has not been called, its state is *Initial*. After an *startService* inter-component method call that targets service A, the state of the component change to *Started*.

Note: the models must be modified in order to take in account external signals. Therefore, in models such as **startActivity**, there must be transitions

from the *Active* state to *Paused* state, and so on so for following the external signals (same as the general model showed in figure 4.7).

```
1  class MainActivity extends Activity {
2      ...
3      public void onStart() {
4        Intent i = new Intent(this, Service1.class);
5        startService(i);
6        ...
7      }
8
9    public void onPause() {
10     Intent i = new Intent(this, Service1.class);
11     stopService(i);
12     ...
13   }
14   ...
15 }
```

Figure 4.16: Activity used to build an AIG

Lets illustrate the algorithm with an example. Figure 4.16 shows the main activity of an application. First, the *startActivity* model (figure 4.7) is built using *MainActivity*. In the **startActivity** model, there is one transition which is that launch task. The analysis inspect the source code of the three callbacks that belong to the launch task and also add edges between this callbacks (following the order in the lifecycle of the particular component that the callback belongs). When the analysis finishes with the launch task, the state of the *MainActivity* is change to *Active*. Moreover, in the launch task, an *startService* method call was found targeting *Service1*. The correspondent model found is built recursively using the component *Service1*. An edge from the state *Active* of *MainActivity* to the state *Initial* of *Service1* and the state of *Service1* is change to *Started*. Then, the next transitions in the model

are executed. When the *pause* action finishes, the state of *MainActivity* is change to *Paused*. Then, since in the callback *onPause* there is the inter-component method call *stopService*, the *stopService* model must be executed on *Service1*–it is worth to mention that there is not any violation of constraints in this example since the state of *Service1* was *Started*. An edge from the state *Paused* state of *MainActivity* to the *Started* state of *Service1* is added. Then, the state of the *Service1* is changed to *Initial*.

After the algorithm finishes its execution, an AIG must be generated. Figure 4.17 shows the resulting AIG. Virtual nodes representing the state of the components are added to reduce the complexity of the analysis.

4.7 Experimental Results

From the experiments, we wanted to evaluate three things: the scalability of building the graph, accuracy in terms of code coverage in the graph against the total number of application's components, and check if any application had violations of model constraints. We ran the implementation of our algorithm against eight applications of the Google market. Table 4.1 shows the general results of the analysis.

Since, the analysis performed to build the AIG is a path-sensitive analysis, we wanted to measure the performance of the analysis. After applying optimizations to the algorithm (the same optimizations applied for the specification miner), we could reach scalable results. For most of the applications it took less than a minute to build the complete graph (see table 4.1). Therefore,

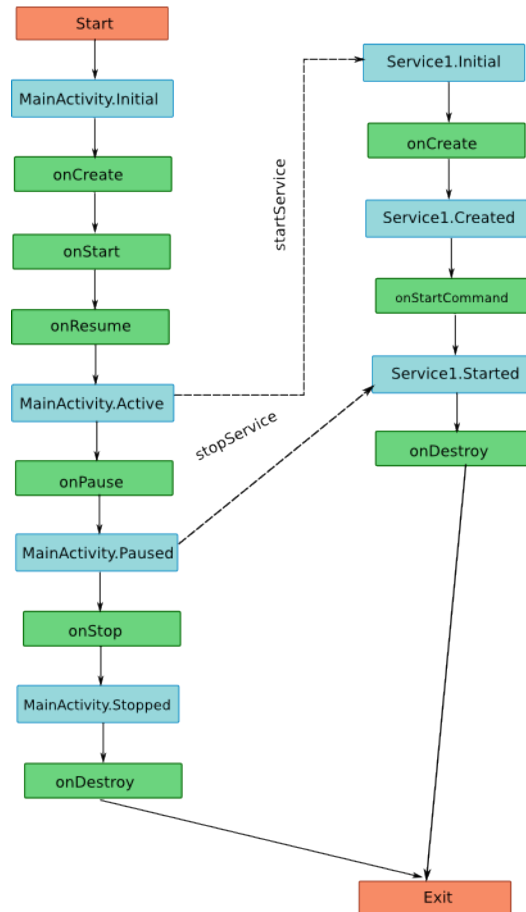


Figure 4.17: Example of an AIG

Table 4.1: Inter-component analysis results

Application	**No. of classes**	**Activities**	**Services**	**Receivers**	**Total**	**Coverage**	**Time (s)**
NoLed 1.5.4	189	2/6	8/10	17/17	27/33	81.00%	70
Sipdroid 2.7	429	7/23	3/3	12/12	23/38	53.00%	396
BLN 1.75	135	4/4	3/3	4/4	11/11	100.00%	30
VLC 2012.011	531	7/14	1/1	3/3	11/18	61.00%	34
MyTracks 1.15	542	4/13	1/1	1/1	5/10	33.00%	38
Skype	1313	1/1	1/1	1/1	3/3	100.00%	34
NYTimes 1.2	394	2/6	1/1	0/1	3/8	38.00%	25

we consider the our algorithm can be used for biggest applications.

In addition, we measure the number of components covered versus the number of total components on the application. The average coverage was of 67%, having over an 80% coverage in three of the eight benchmarks. The results show that our algorithm had a good coverage on services and broadcast receivers having an average coverage of 95%, whereas, activities had a lower percentage of coverage. We attribute the main cause of false negatives to that in some cases the analysis could not identify which components were supposed to be activated in the inter-component method call (the *Intent* object was not identify). In some cases, applications use helper functions to create the intents. For example, the class *org.sipdroid.sipua.ui.Receiver* in *Sipdroid 2.7* uses a helper function *createIntent* to generate all the intents to start activities (see figure 4.18). This causes our analysis, an intra-procedural value flow, to not find the intent used in the inter-component call.

```
1 static Intent createIntent(Class<?>cls) {
2   Intent startActivity = new Intent();
3   startActivity.setClass(mContext, cls);
4   startActivity.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
5   return startActivity;
6 }
```

Figure 4.18: Sipdroid 5.7 helper function to create intents

Another aspect that the analysis checked during building the graph, was whether there were any kind of violations of the model constraints defined in the section 4.4. One of these constraints was that broadcast receivers cannot *bind* or *unbind* services (see section 4.4.2). The official documentation

emphatically mentions that services can be bound just by services, activities and content providers [4]– failing this constraint can add non-deterministic behavior to the application. We found that *DroidNotify* contains method calls *bindService* and *unbindService* inside a broadcast receiver, violating this constraint.

4.8 Conclusion

In conclusion, in this chapter an inter-component model used to generate an intermediate representation of Android applications is presented. Moreover, we implemented an algorithm that generates an *AIG* of Android applications and run it against eight applications from the Google market. The results show that the analysis has high coverage of the whole application when the intent objects are well recognized.

Chapter 5

Conclusions and Future Work

In conclusion, two approaches to model Android applications are expected to be developed: *Mining Energy Specifications (E-Specs) and Energy Bugs (E-Bugs)* and *Building a Representation for Analyzing Android Applications.* The first approach uses the Android components lifecycles to define a control flow of Android applications. We test this approach applying a specification miner to check if the representation is enough to find energy related bugs. We proved our hypothesis that inter-component interactions between components have impact in the control flow of Android applications and that just taking the Android components lifecycles is not enough to analyze Android applications. That is why, in the second project a representation of Android applications is defined taking in account both, Android components lifecycles and the interaction between components. We studied the behavior of Android applications in order to model the system. We use this model to generate an Android Inter-component Control Flow Graph of Android applications. Our implementation had 67% of code coverage against eight applications from the Google Market and 100% on two of them.

The results for both, specification miner and the AIG builder, are lim-

ited to in-accuracies of concurrency code and aliasing. Therefore, thread analysis and a better value flow and/or points to analysis can be added to improve the results. Moreover, the graph representation developed in chapter 4 can be used for other analysis including specification mining and also for testing.

Bibliography

[1] Android activities. http://developer.android.com/guide/components/activities.html.

[2] Android developers - android activations. https://plus.google.com/108967384991768947849/posts/jHLD6HTfx9U.

[3] Android fundamentals. http://developer.android.com/guide/components/fundamentals.html.

[4] Android services. http://developer.android.com/guide/components/services.html.

[5] Android tasks and back stack. http://developer.android.com/guide/components/tasks-and-back-stack.html.

[6] Application programming interface. http://en.wikipedia.org/wiki/Application_programming_interface.

[7] dex2jar - tools to work with android .dex and java .class filess. http://code.google.com/p/dex2jar/.

[8] Email application partial wake lock. https://code.google.com/p/android/issues/detail?id=9307.

[9] Findbugs. http://findbugs.sourceforge.net/.

[10] Jlint. http://jlint.sourceforge.net/.

[11] Number of available android applications. http://www.appbrain.com/stats/number-of-android-apps.

[12] Pending intents. http://developer.android.com/reference/android/app/PendingIntent.html.

[13] Moutaz Alazab, Veelasha Monsamy, Lynn Batten, Patrik Lantz, and Ronghua Tian. Analysis of malicious and benign android applications. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 608–616. IEEE, 2012.

[14] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[15] Glenn Ammons, Rastislav Bodík, and James R Larus. Mining specifications. In *ACM Sigplan Notices*, volume 37, pages 4–16. ACM, 2002.

[16] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[17] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM*

SIGCOMM conference on Internet measurement conference, pages 280–293. ACM, 2009.

[18] Rastisalv Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–251. ACM, 1998.

[19] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[20] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, volume 17, pages 18–25, 2012.

[21] Patrick PF Chan, Lucas CK Hui, and SM Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136. ACM, 2012.

[22] Avik Chaudhuri. Language-based security on android. In *Proceedings of the ACM SIGPLAN fourth workshop on programming languages and analysis for security*, pages 1–7. ACM, 2009.

[23] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[24] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[25] Thomas Wagner Peter Wolstenholme Ferdinand Wagner, Ruedi Schmuki. *Modeling Software with Finite State Machines: A Practical Approach.* Auerbach Publications, 2006.

[26] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.

[27] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 339–349. ACM, 2008.

[28] Vinod Ganapathy, David King, Trent Jaeger, and Somesh Jha. Mining security-sensitive operations in legacy code using concept analysis. In

Proceedings of the 29th international conference on Software Engineering, pages 458–467. IEEE Computer Society, 2007.

[29] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[30] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[31] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Estimating android applications' cpu energy usage via bytecode profiling. In *Green and Sustainable Software (GREENS), 2012 First International Workshop on*, pages 1–7. IEEE, 2012.

[32] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 35th International Conference on Software Engineering*, 2013.

[33] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android

applications. In *Proceedings of the Workshop on Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, 2012.

[34] Zhifeng Lai, SC Cheung, and WK Chan. Inter-context control-flow and data-flow test adequacy criteria for nesc applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 94–104. ACM, 2008.

[35] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for java. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 73–79. ACM, 2001.

[36] David Lo and Siau-Cheng Khoo. Quark: Empirical assessment of automaton-based specification miners. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 51–60. IEEE, 2006.

[37] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[38] Zheng Lu and Supratik Mukhopadhyay. Model-based static source code analysis of java programs with applications to android security. In *Com-*

puter Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual, pages 322–327. IEEE, 2012.

[39] Atif M Memon, Mary Lou Soffa, and Martha E Pollack. Coverage criteria for gui testing. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 256–267. ACM, 2001.

[40] Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

[41] Nguyet Nguyen and Mary Lou Soffa. Program representations for testing wireless sensor network applications. In *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*, pages 20–26. ACM, 2007.

[42] Abhinav Pathak, Y. Charlie Hu, and Ming 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*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[43] Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel 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*, pages 267–280. ACM, 2012.

[44] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel 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*, MobiSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[45] Dewayne E Perry and W Michael Evangelist. An empirical study of software interface faultsan update. In *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, volume 2, pages 113–126, 1987.

[46] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. *ACM SIGPLAN Notices*, 42(6):123–134, 2007.

[47] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 224–238. IEEE, 2012.

[48] Android Sensor. `http://developer.android.com/reference/android/hardware/SensorManager.html`.

[49] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333. IEEE, 2010.

[50] Sharon Shoham, Eran Yahav, Stephen J Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. *Software Engineering, IEEE Transactions on*, 34(5):651–666, 2008.

[51] Nishant Sinha and Chao Wang. Staged concurrent program analysis. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 47–56. ACM, 2010.

[52] Stefan Staiger. Static analysis of programs with graphical user interface. In *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, pages 252–264. IEEE, 2007.

[53] Mirko Streckenbach and Gregor Snelting. Points-to for java: A general framework and an empirical comparison. Technical report, Technical report, U. Passau, 2000.

[54] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.

[55] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

[56] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. Towards verifying android apps for the absence of no-sleep energy bugs. In

Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems, pages 3–3. USENIX Association, 2012.

[57] Westley Weimer and George C Necula. Mining temporal specifications for error detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476. Springer, 2005.

[58] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[59] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal api rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*, pages 282–291. ACM, 2006.

[60] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Software Engineering (WCSE), 2012 Third World Congress on*, pages 101–104. IEEE, 2012.

[61] Zhibo Zhao and Fernando C Colon Osono. trustdroid: Preventing the use of smartphones for information leaking in corporate networks through the used of static analysis taint tracking. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 135–143. IEEE, 2012.