The construction and applications of callback control flow graphs for event-driven and framework-based mobile apps

Danilo Dominguez Perez

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Dominguez Perez, Danilo, "The construction and applications of callback control flow graphs for event-driven and framework-based mobile apps" (2019). Graduate Theses and Dissertations. 17004.
https://lib.dr.iastate.edu/etd/17004

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
The construction and applications of callback control flow graphs for event-driven and framework-based mobile apps

by

Danilo Dominguez Perez

A dissertation submitted to the graduate faculty in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Wei Le, Major Professor
Hridesh Rajan
Samik Basu
Robyn Lutz
Wensheng Zhang

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University
Ames, Iowa
2019

Copyright © Danilo Dominguez Perez, 2019. All rights reserved.
DEDICATION

I would like to dedicate this thesis to my son Evan and my wife Dayiris without whose support I would not have been able to complete this work. Dayiris, you supported my dream and were always for me every time I needed you. I would also like to thank my friends and family for their loving guidance during the writing of this work.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>CHAPTER 1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Contributions</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Outline</td>
<td>4</td>
</tr>
<tr>
<td>CHAPTER 2. PREDICATE CALLBACK SUMMARIES</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Defining PCS</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Computing PCS</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Identifying Callback Nodes</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Computing Predicate Nodes</td>
<td>11</td>
</tr>
<tr>
<td>2.2.3 Computing Update Nodes</td>
<td>14</td>
</tr>
<tr>
<td>2.2.4 Generating Summary Graphs</td>
<td>15</td>
</tr>
<tr>
<td>2.3 Applying PCS</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Experimental Results</td>
<td>17</td>
</tr>
<tr>
<td>2.4.1 Experimental Setup</td>
<td>17</td>
</tr>
<tr>
<td>2.4.2 Compactness of the Summaries</td>
<td>18</td>
</tr>
<tr>
<td>2.4.3 Correctness of the Summaries</td>
<td>19</td>
</tr>
<tr>
<td>2.4.4 Scalability of Generating the Summaries</td>
<td>20</td>
</tr>
<tr>
<td>2.4.5 Constructing Apps’ inter-callback CFGs with PCSs</td>
<td>21</td>
</tr>
</tbody>
</table>
CHAPTER 3. SPECIFYING CALLBACK CONTROL FLOW OF ANDROID APPS USING FINITE AUTOMATA

3.1 An Overview

3.1.1 What is a CCFA

3.1.2 How to Compute a CCFA

3.2 Callback Control Flow Automata

3.2.1 Definition

3.2.2 The Scope of CCFAs

3.3 Computing a CCFA

3.4 Program Analysis using CCFAs

3.4.1 First Approach

3.4.2 Second Approach

3.5 Evaluation

3.5.1 Implementation and Experimental Setup

3.5.2 Performance

3.5.3 Callback Coverage

3.5.4 Information Flow Analysis

3.5.5 Value-Flow Analysis for Source-Sink Pairs

3.6 Discussions

3.7 Summary

CHAPTER 4. TESTING COVERAGE CRITERIA FOR MOBILE APPS BASED ON CALLBACK SEQUENCES

4.1 Motivation

4.2 Designing Test Criteria Based on Callback Interactions
LIST OF TABLES

| Table 2.1 | Matching Predicate Nodes and Update Nodes | 14 |
| Table 2.2 | Size of ICFGs versus Size of Summary Graphs | 19 |
| Table 2.3 | Comparison to Manually Identified Ground Truths | 20 |
| Table 2.4 | Constructing Apps’ inter-callback ICFGs Using PCSs | 21 |
| Table 2.5 | Compare Dynamic Traces and Static Paths | 22 |
| Table 3.1 | Performance of Computing CCFAs | 46 |
| Table 3.2 | FLOWDROID on lifecycle model and on CCFA | 48 |
| Table 3.3 | Information Flow Analysis on DroidBench | 50 |
| Table 3.4 | Global Source Sink Pairs | 52 |
| Table 4.1 | Dataflow between callbacks. EV: event callback, AS: API_SYNC callback, AA: API_ASYNC callback | 60 |
| Table 4.2 | Bug study results. EV: event callback, AS: API_SYNC callback, AA: API_ASYNC callback | 61 |
| Table 4.3 | Benchmark Apps | 73 |
| Table 4.4 | The bugs found and their correlations with the increases of the test criteria | 76 |
| Table 4.5 | Unique crashes found by both tests, by just manual testing, or by just Monkey | 77 |
| Table 4.6 | Coverage of C1, C2 and C3 using manual testing and Monkey | 78 |
| Table 4.7 | Compare the time used to trigger the bugs | 78 |
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>Source code for <code>LoaderManager.initLoader</code></td>
<td>7</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>An Overview of Lithium</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td><code>handleMessage</code> in <code>ActivityThreadHandler</code></td>
<td>10</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Scalability of Lithium</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Source code, CCFA, WTG and PCS of an example app</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>An Overview of the Approach</td>
<td>28</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Guard for <code>sendMessage</code></td>
<td>32</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Motivating examples using real world apps chanu and FileDownloader. We present a fragment of the code that is significant to the bug and the possible sequences of callbacks in the fragment that can be executed at runtime</td>
<td>58</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>An app with Location Callbacks</td>
<td>63</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>The Framework for Measuring Coverage</td>
<td>66</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>The CCFA for the code shown in Figure 4.1a</td>
<td>70</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Coverage over time for Pedometer and Movie DB</td>
<td>75</td>
</tr>
</tbody>
</table>
I would like to thank my advisor Wei Le for her support and guidance to help me become a researcher. Professor Le was always an example of hard work and dedication. These values were imprinted in me during my studies. Without her feedback and countless hours of meetings, this dissertation would not have been possible. I would like to also thank my committee members for their feedback and support, not just technical, but emotional during my studies: Hridesh Rajan, Robyn Lutz, Samik Basu and Wensheng Zhang.

I would like to thank all the lab mates I have had during my studies. Their camaraderie in the lab always kept an environment that encourage hard work and challenging discussions. Their support help me stayed in the lab countless hours during deadlines.

My graduate studies would not be possible without the support of the government of Panama through their Senacyt-IFARHU scholarships program. This program provided full support for me and my family.
Mobile devices have become ubiquitous over the last years. Android, as the leading platform in the mobile ecosystem, have over 2.5 million apps published in Google Play Market. This enormous ecosystem creates a fierce competition between apps with similar functionality in which the low quality of apps has been shown to increase the churn rate considerably. Additionally, the complex event-driven, framework-based architecture that developers use to implement apps imposes several challenges and led to new varieties of code smells and bugs. There is a need for tools that assure the quality of apps such as program analysis and testing tools. One of the foundational challenges for developing these tools is the sequencing or ordering of callback methods invoked from external events (e.g. GUI events) and framework calls. Even for a small subset of callbacks, it has been shown that the current state-of-the-art tools fail to generate sequences of callbacks that match the runtime behavior of Android apps.

This thesis explores the construction and applications of new representations and program analyses for event-driven, framework-based mobile applications, specifically Android apps. In Android, we observe that the changes of control flow between entry points are mostly handled by the framework using callbacks. These callbacks can be executed synchronously and asynchronously when an external event happens (e.g. a click event) or a framework call is made. In framework-based systems, method calls to the framework can invoke sequences of callbacks. With the high overhead introduced by libraries such as the Android framework, most current tools for the analysis of Android apps have opted to skip the analysis of these libraries. Thus, these analyses missed the correct order of callbacks for each callback invoked in framework calls. This thesis presents a new specification called Predicate Callback Summary (PCS) to summarize how library or API methods invoke callbacks. PCSs enable inter-procedural analysis for Android apps without the overhead of analyzing the whole framework and help developers understand how their code (callback methods)
is executed in the framework. We show that our static analysis techniques to summarize PCSs is accurate and scalable, considering the complexity of the millions of lines of code in the Android framework.

With PCSs summaries, we have information about the control flow of callbacks invoked in framework calls but lack information about how external events can execute callbacks. To integrate event-driven control flow behavior with control behavior generated from framework calls, we designed a novel program representation, namely Callback Control Flow Automata (CCFA). The design of CCFA is based on the Extended Finite State Machine (EFSM) model, which extends the Finite State Machine (FSM) by labeling transitions using information such as guards. In a CCFA, a state represents whether the execution path enters or exits a callback. The transition from one state to another represents the transfer of control flow between callbacks. We present an analysis to automatically construct CCFAs by combining two callback control flow representations developed from the previous research, namely, Window Transition Graphs (WTGs) and PCSs. To demonstrate the usefulness of our representation, we integrated CCFAs into two client analyses: a taint analysis using FLOWDROID, and a value-flow analysis that computes source and sink pairs of a program. Our evaluation shows that we can compute CCFAs efficiently and that CCFAs improved the callback coverages over WTGs. As a result of using CCFAs, we obtained 33 more true positive security leaks than FLOWDROID over a total of 55 apps we have run. With a low false positive rate, we found that 22.76% of source-sink pairs we computed are located in different callbacks and that 31 out of 55 apps contain source-sink pairs spreading across components.

In the last part of this thesis, we use the CCFAs to develop a new family of coverage criteria based on callback sequences for more effective testing Android apps. We present 2 studies to help us identify what types of callbacks are important when detecting bugs. With the help of the empirical results, we defined 3 coverage criteria based on callback sequences. Our evaluation shows that our coverage criteria are a more effective metric than statement and GUI-based event coverage to guide test input generation.
CHAPTER 1. INTRODUCTION

Smartphones and tablets have become ubiquitous devices recently, even overtaking desktop and laptop computers as the main medium to access the Internet. Android is the leading mobile platform with over 1.4 billion Android devices sold [17] and more than 2.5 million apps published on the official Google Play Store [57]. Nonetheless, the complex event-driven, framework-based architecture that developers use to implement apps imposes some challenges on the quality of Android apps and led to new varieties of code smells and bugs [44, 48, 31, 40]. Due to the impact of Android apps on the market and the complexity of developing apps, there is a need for programming tools that analyze Android apps. One of the foundational challenges for developing these tools is the sequencing or ordering of callback methods invoked from external events (e.g. GUI events) and framework calls. Even for a small subset of callbacks, it has been shown that the current state-of-the-art tools fail to generate sequences of callbacks that match the runtime behavior of Android apps [63].

In this regard, previous work soundly models the ordering of callbacks by computing their product graph [38]. Analyzing the product graph of callbacks can become intractable with apps that reach a certain size, and given the constraints of the system semantics, a great portion of the callback paths generated are infeasible. State-of-the-art analyses use techniques, such as the lifecycle of components [26], to model a limited number of control-flow constraints between callbacks. However, besides components, several features of the Android framework are implemented through callbacks. Therefore, current techniques miss the correct ordering of these component’s callback. Moreover, some of these tools do not correctly model callback orderings across two different activities [63]. There are other representations such as Static Window Transitions Graphs (WTG) are more precise regarding interleaving of some callbacks, but focus on a small subset of callbacks including GUI
entities such as activities, dialogs, views, and widgets, covering just a small portion of all the callbacks defined by the apps.

As we mentioned, framework calls can also invoke sequences of callbacks. Current analyses of Android apps skip the overhead of analyzing libraries such as the Android framework and built application-only call graphs, missing vital information, including changes of control to callbacks invoked from calls to the framework (API calls)—e.g. the API method `startService` invokes several callbacks of a service object passed as parameter to the call. However, we found that calls to framework methods can asynchronously (using an event-queue mechanism) or synchronously invoke multiple callbacks.

This thesis introduces a summary representation for control flow of callbacks. We proposed a representation, called *predicate callback summary (PCS)*, and static analysis techniques that automatically compute the PCSs from the Android API methods [49]. A PCS is a control flow-based graph representation that extracts from the API implementation 1) all the callback sequences, 2) the predicates depending on which the callbacks are executed, and 3) the updates that can determine the outcome of the predicates. The callback sequences in PCSs enable inter-procedural analysis for Android apps and help developers understand how their code (callback methods) is executed in the framework. The predicate and update nodes in the PCSs are used to determine the feasibility of program paths, which are important for eliminating false positives in static analysis and for improving the evaluation of test coverage. The predicate nodes are also useful for generating test inputs that can exercise interesting callback sequences, e.g., testing the interface between the app and the Android framework.

To integrate the sequence of callbacks generated from an external event and from framework calls, we designed a novel program representation, namely *Callback Control Flow Automata (CCFA)* [50]. The design of CCFA is based on the *Extended Finite State Machine (EFSM)* [51] model, which extends the Finite State Machine (FSM) by labeling transitions using information such as *guards*. In a CCFA a state represents whether the execution path enters or exits a callback. The transition from one state to another represents the transfer of control flow between callbacks.
This representation is then used as the base for different interprocedural static analyses. In the first approach, we extend an existing inter-procedural, context-sensitive analysis to inter-callback dataflow analysis using CCFAS, and also instantiate it to compute inter-callback source and sink pairs of a program. In the second approach, we generate program stubs using the callback invocation paths provided by CCFAS to directly integrate CCFAS with any other static analysis tools for Android.

Another application of the CCFAs this thesis introduces is the design and implementation of coverage criteria based on the execution of callbacks. We consider coverage criteria based on callback sequences because they are able to cover behaviors that occur between different components of the mobile phone. For instance, we can recognize different behaviors by covering callback interleavings between event handlers (including any kind of event such as GUI, location, etc.), asynchronous and synchronous callbacks invoked in API methods. Thus, a coverage criterion based on different kind of callbacks can cover behaviors such as the execution of background tasks using Services, an advantage over black-box coverage criteria. We use the CCFA to identify paths of callbacks that should be tested.

1.1 Contributions

This thesis makes the following contributions:

- we define a novel specification, PCS, for summarizing control flow of callbacks in a library method,

- we present static analysis techniques and a tool, Lithium, to compute such summaries from complex libraries such as the Android framework,

- we define the callback control flow automata CCFAs, a control flow representation for event-driven, framework-dependent applications such as Android apps,

- we design the algorithm to automatically compute such representation using apps and framework source code,
• we implemented the tools for constructions of CCFAs and PCSs, and experimentally demonstrated that our algorithms are scalable and practical,

• we designed and implemented new coverage criteria based on sequences of callbacks, and

• we present an empirical evaluation of the new coverage criteria and show their importance to verify the effectiveness of tests for Android apps

1.2 Outline

The remainder of this thesis is organized as follows. Chapter 2 presents the design and implementation of Predicate Callback Summaries. We evaluate our algorithms against the Android framework 4.1. Chapter 3 presents the definition and algorithms to build Callback Control Flow Automata. We show how this representation is used to improve results over state-of-art tools. In Chapter 4 presents new coverage criteria for testing Android apps. We present related work in Chapter 5, and the conclusions and future work in Chapter 6.
CHAPTER 2. PREDICATE CALLBACK SUMMARIES

This chapter presents a specification technique, called predicate callback summary (PCS), and static analysis techniques that automatically compute the PCSs from the Android API methods. A PCS is a control flow–based graph representation that extracts from the API implementation 1) all the callback sequences, 2) the predicates depending on which the callbacks are executed, and 3) the updates that can determine the outcome of the predicates. The callback sequences in PCSs enable inter-procedural analysis for Android apps and help developers understand how their code (callback methods) is executed in the framework. The predicate and update nodes in the PCSs are used to determine the feasibility of program paths, which are very important for eliminating false positives in static analysis and for improving the evaluation of test coverage. The predicate nodes are also useful for generating test inputs that can exercise interesting callback sequences, e.g., testing the interface between the app and the Android framework.

2.1 Defining PCS

A Predicate Callback Summary (PCS) is an interprocedural abstraction for library methods (in our case for methods of the Android API) that keep the control flow information about the callbacks that are invoked and the context needed for their invocation. More formally, a PCS is a directed graph $G = (N_c \cup N_p \cup N_u, E)$ where $N_c$ is the set of callback nodes, $N_p$ is the set of predicate nodes and $N_u$ is the set of statements which update values of variables used in predicate nodes; called update nodes. $E$ is the set of edges between any two nodes in the set $N_c \cup N_p \cup N_u$. PCSs aim to summarize all potential execution paths of the callbacks implemented in library methods. In the rest of the section, we explain each type of node.

Callback Nodes. The framework executes callback methods through objects passed from the app that are instances of classes extending the Android API. There are two ways an app can pass
the object receiver of a callback to the framework. Commonly, the object receiver can be passed through the parameters (including fields of the parameters) of an API call. The object receiver can also be the calling object of an API call. For example, when the object receiver is an instance of a class extended by the app and the API call executes a method that was overridden by the app. A callback node $n \in N_c$ represents a callback call site that is executed in an API method. The two key pieces of information specified in callback nodes are 1) the object receiver of the callback, and 2) the callback signature. This information is useful to identify the correct callback methods in the app during client analysis.

**Predicate Nodes.** A predicate node $n \in N_p$ provides the conditions that need to be satisfied for the execution of a callback or a sequence of callbacks. They are useful, along with update nodes, to determine the feasibility of a sequence of callbacks to be executed at an API call site.

The predicate nodes are abstracted to boolean expressions of type $a \ op \ b$ where $\ op \in \{<, >, \leq, \geq, ==, !=\}$ and $a$ and $b$ can be constants, abstract variables or arithmetic expressions on constants and abstract variables. For instance, in Figure 3.1, $g.\text{thread} == \text{null}$ at node 2 in `startService` and $g.\text{started} != \text{true}$ at node 3 in `unbindService` are predicate nodes. In these two boolean expressions, the left operand is abstracted to abstract variables ($\text{static, g, thread}$) and ($\text{static, g, started}$), and the right operands are resolved to constants (null and boolean constant respectively).

**Update Nodes.** Once we have all the predicate nodes, we need to find all the statements that can affect the outcomes of these predicates. Statically, an update node $n \in N_u$ is an assignment whose left side is a variable used in one of the predicate nodes found. These assignments can contribute completely or partially to change the outcomes of predicate nodes. Dynamically, the update nodes are the ones that can change the program state and impact the invocations of succeeding callbacks located in the current API method or the succeeding API methods.

For example, the Android framework keeps a map of all the `Service` objects running in an app. The map is created when the app starts, and it can be modified and accessed throughout the app’s lifetime using a static variable defined on the framework. When any `Service` object is going to
be started, the app invokes `startService`. This API call first checks if the service object is already started by inquiring the static variable. If the object does not exist, the app creates the corresponding service by calling the `onCreate` callback and also updates the state of the service stored in the static variable. In this case, predicate conditions on the static variables are predicate nodes and the statements that change its values are update nodes.

```java
class LoaderManager {
    Loader<D> initLoader(int id, Bundle args, LoaderCallbacks<D> c) {
        LoaderManager r0 = this;
        boolean creatingLoader = r0.mCreatingLoader
        if (creatingLoader == true) {
            throw new IllegalStateException("...");
        }
        LoaderInfo info = r0.mLoaders.get(id);
        if (info == null) {
            info = r0.oldLoader;
            r0.mLoaders.put(id, info);
            c.onCreateLoader(); // callback call
        }
        boolean haveData = info.mHaveData;
        reset(haveData);
    }
    void reset(boolean haveData, LoaderCallbacks<D> c) {
        if (haveData == true) {
            c.onReset();
        }
    }
}
```

Figure 2.1: Source code for `LoaderManager.initLoader`

One of the requirements for the PCSs is that they need to be directly plugged in at any API call site and analyzed with apps’ code without any re-computation of the API method. This requires that the variables and expressions used in the three types of nodes are also visible to the client or to the other API methods invoked by the client. For that, we need an abstraction for the local variables used in predicates. In our analysis, each local variable is resolved to a set of aliases represented by access paths [56]. The access paths are resolved in a backward analysis from the predicate node until a visible variable (they are either the calling object of the API method, a parameter of the API method or a static variable) is obtained.

For example, see the code snippet of the API method `initLoader` in Figure 2.1. At line 19, the callback `onReset` is invoked through the object receiver `c`, with predicate `haveData ==`
true (see line 18). However, haveData is a local variable and is not visible outside the API method initLoader. Presenting such information in the PCS does not help to determine whether haveData is true and whether the callback onReset() will be invoked at the API call site. Therefore, in the PCS, we map the local variable haveData to an abstract variable. For instance, along with the path that traverses lines 4, 10 and 14, we obtain an abstract variable represented using a three-tuple (calling object, LoaderManager, oldLoader.mHaveData). The tuple indicates that the variable is computed from the calling object of the API method, with the class type LoaderManager, and we can compute it by first accessing its field oldLoader and then mHaveData.

Generally, in the three-tuple that specifies the abstract variable, the first element specifies the visibility and scope of the variable and has the domain of static variable (from the classes of the Android framework), the API calling object or the input parameters of the API method. The second element of an abstract variable specifies the class type of the variable. The third element provides the details on how to compute the values for the variable, e.g., via an access path.

2.2 Computing PCS

In this section, we present our algorithms to compute PCSs from the source code of Android API methods. Figure 2.2 shows our static analysis framework, Lithium, which have 4 main phases: identify callback nodes, compute predicate nodes, compute update nodes, and generate the summary graphs. The framework uses ICFGs generated from the source code of the API methods as input.

2.2.1 Identifying Callback Nodes

The goal of this phase is to locate callback call sites in API methods. Specifically, we aim to identify the program paths that can reach the callbacks from the entry of the API method. Along these paths, we then can generate summaries for predicate nodes and update nodes.
2.2.1.1 Searching for Callback Call Sites

In the first step, we address the question of which calls invoked in an API method can be callbacks? We identify a list of potential callback signatures by analyzing the class hierarchy and the visibility of the methods in the Android framework. Specifically, we find every non-static, non-final method of a non-final class that has a visibility of public or protected. Additionally, we also identify all the methods of public interfaces in the Android framework. These methods can be overridden by the apps and executed in the Android framework via dynamic dispatch and thus they can be callbacks.

Once we have the list of potential callback signatures, we inspect all the call sites in the framework and verify if they match one of the callback signatures. If a callback signature is matched, we perform a backward traversal from the call site on the call graph until an API method is reached. This step can generate a set of call chains of the form \( \{m_0, ..., m_n\} \) where \( m_0 \) is an API method followed by a sequence of framework method calls until the callback \( m_n \) is reached. We define \( C \) to be the set of all the call sites that match a callback signature and the function \( \text{Chains}(c) \), for a given a call site \( c \in C \), return the set of all call chains found that reach \( c \). We use these call chains in the next phases of our analysis to ensure that it only traverses the paths of interest for scalability.
Handling Asynchronous Callbacks  One of the challenges to identify the call chains is that there are callbacks that are invoked implicitly through a message passing mechanism developed in Android[30]. The message passing mechanism is implemented using the Handler class on the framework through a pair of methods sendMessage and handleMessage. At runtime, when sendMessage is invoked from a Handler object in the API method, a message is posted into an event queue. When the message is dispatched, handleMessage from the same Handler class is invoked to handle the message. The handleMessage method can invoke a set of callbacks.

To include such asynchronous callbacks into the summary, we need to construct an edge to connect sendMessage and its corresponding handleMessage in the ICFG. To do so, we first determine the class name (type) of the object receiver for every asynchronous call of sendMessage. We then find the handleMessage method defined in the class. For example, in Figure 2.3, according to line 10, the object receiver of sendMessage at line 12 has a type ActivityThreadHandler. Thus, we identify that handleMessage, implemented in the class ActivityThreadHandler at line 2, is a match and we create an implicit edge from sendMessage to handleMessage in the framework’s call graph. Any callback invoked in this method should be linked to the asynchronous call site of sendMessage at line 12.

```java
1 class ActivityThreadHandler extends Handler {
2     public void handleMessage(Message m) {
3         switch (m.what) {
4             case LAUNCH_ACTIVITY:
5                 handleStartActivity(...); break;
6         }
7     }
8 }
9 class Activity {
10     ActivityThreadHandler threadHandler;
11     public void startActivityForResult(...) {
12         threadHandler.sendMessage(LAUNCH_ACTIVITY);
13     }
14 }
```

Figure 2.3: handleMessage in ActivityThreadHandler
2.2.1.2 Resolving Object Receivers

In this step, we identify the object receiver of the call sites found in the previous step. Algorithm 1 shows the steps to resolve the object receivers for each call chain and returns the set of callback nodes $N_c$. The procedure takes as input the set of all the call sites that match a callback signature $C$ and the ICFG generated for the framework. At line 3 we find aliases for the base object of the call sites using a demand-driven alias analysis used in [8]. At line 6 we verify if the base object of the call site (first element in call chain) is an alias of the caller object in the API method. At line 7 we add the this (caller) reference to the receivers. Then, we verify the parameters of the API call. The function $\text{Params}$ returns the parameters references of the API call. We evaluate each parameter and its fields (in case they may point to the object receiver of the callback) using the function $\text{MatchParam}$. We use $\text{Unknown}$ for the case when the object receiver is not resolved to the caller or a parameter. This can happen when the object receiver is passed through a different API method and is stored in the framework. For example, the method $\text{TextView.setText}$ executes callbacks for objects of type $\text{TextWatcher}$ which are stored in a list of listeners updated by the API method $\text{TextView.addTextChangedListener}$. There is also the case when the object receiver is internal to the framework and it cannot point an object passed from the app, which means it cannot be a callback method. We leave the detection of such false positives for future work.

2.2.2 Computing Predicate Nodes

Our approach here is to first perform control flow analysis to identify conditional branches that a callback node is transitively control dependent on, and report them as predicate nodes (Section 2.2.2.1). We then resolve the local variables contained in the predicate nodes to symbolic expressions of abstract variables via a backward symbolic substitution (Section 2.2.2.2).

2.2.2.1 Identifying Predicate Nodes

Predicate nodes are the conditional branch statements in the ICFG of the API method that decides whether a callback should be executed. For each method $m_i$ appeared in the call chain
{m_0, ..., m_n} (computed in the phase Identifying Callback Nodes), we applied a control flow analysis shown in Algorithm 2.

The inputs of the algorithm are the CFG of the method m_i, and the program point of interest p. Depending on m_i, p can either be the callback c or the call site of the method m_{i+1} in the call chain. The algorithm reports a set of branch nodes which p is transitively control dependent on. At line 2, the algorithm traverses every conditional branch in the method. At line 3, Influence(b) returns all the statements in the CFG that are transitively control dependent on the conditional branch statement b [64] [65]. If the statement of interest p is one of such statements, the branch node b will be stored in the results B.
2.2.2.2 Summarizing Predicate Nodes

The goal of this step is to convert any of the local variables in the predicate nodes to be *abstract expressions*. The approach we used is demand-driven, symbolic back-substitution [14]. We unroll loops once to assure termination of the back-substitution algorithm. The analysis starts at each predicate node and propagates backward along all paths reachable from the predicate nodes. At any assignment that defines the local variables under tracking, we update the variables symbolically. For local variables, this substitution can generate a set of access paths. The analysis ends when the variable is resolved to an expression of abstract variables or constants, or when the traversal reaches the entry of the API method. We use these abstract variables to identify update nodes (see Section 2.2.3).

As an example, in Figure 2.1, the local variable *haveData* in the predicate at line 18 can be resolved to \{\((\text{calling object, LoaderManager, oldLoader.mHaveData}) \lor (\text{calling object, LoaderManager, mLoaders.get().mHaveData})\}\}, where \(\lor\) indicates the merge of the abstract variables along the path at lines 14, 10 and 4, and along the path at lines 14, 8 and 4.

From the above example, we show that at each statement determined to be relevant to the variable under tracking, we update the symbolic expressions. In case of a field dereference, we add the field to the access path under tracking and keep inquiring whether the base object can be resolved to static variables, input parameters or calling objects. Due to the unprohibited expenses, our current implementation does not traverse to callee methods when it reaches a method call and treats them as if they were fields, adding them to the computed access path together with field references (see the above example `mLoaders.get().mHaveData`). Our insight is that the expressions on abstract variables in the predicate nodes are used together with the update nodes to decide infeasible paths, and in our empirical studies, we found that among one the most common cases where local variables were computed to return values, the methods belonged to collection classes (*JDK* and Android collections); therefore, we manually define the templates for these classes given their side effects, and use them to resolve the correlations between predicate and update nodes for infeasible paths.
In Table 2.1, we show a few examples of the templates we developed. Under *Class*, we show to which class the methods belong to. Under *Predicate Nodes*, we list the method calls frequently used in predicate nodes. The column provides a set of methods used to test the state of a collection. For example, `isEmpty` tests if a collection is empty. Under *Update Nodes*, we list the types of statements that potentially can answer questions regarding the conditions in the predicate node—these are the methods that have side effects on the collections. Note that the methods whose names end with a star (*) represent a set of similar methods whose names start with the same prefix. For example, we have the method `contains` to test if a member belongs to a collection, and `containsAll` tests if a set of values belong to a collection. We represent such calls using `contain*`.

### 2.2.3 Computing Update Nodes

The goal of including update nodes in the PCSs is to determine under a particular condition, which callbacks (either in the current API method or in the succeeding API methods) should be invoked. To determine which statements in the API method can be update nodes, we need to 1) obtain all the abstract variables used in the predicate nodes for all the Android API methods, and 2) find the assignments and the framework calls that can potentially change the values of these variables.

<table>
<thead>
<tr>
<th>Class</th>
<th>Predicate Nodes</th>
<th>Update Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.List</td>
<td><code>isEmpty, size, get*, contains*</code></td>
<td><code>add*, remove*, set</code></td>
</tr>
<tr>
<td>java.util.Set</td>
<td><code>isEmpty, size, contains*</code></td>
<td><code>add*, remove*</code></td>
</tr>
<tr>
<td>java.util.Map</td>
<td><code>isEmpty, size, contains*, get</code></td>
<td><code>put*, remove</code></td>
</tr>
<tr>
<td>android.util.ArrayMap</td>
<td><code>isEmpty, size, value*, contains*</code></td>
<td><code>setValueAt, put*, remove*</code></td>
</tr>
<tr>
<td>android.util.SparseArray</td>
<td><code>size, value*</code></td>
<td><code>setValueAt, put*, remove*, delete</code></td>
</tr>
</tbody>
</table>
We start by analyzing assignment statements in the API methods and summarizing the destination of the assignments into abstract variables and their expressions. We terminate the analysis for this assignment early if we found that the access path we obtained so far does not match any of the abstract variables in the list. In the second step, we examine the method calls to identify possible matches with templates defined in Table 2.1. When a method call signature matches a template, we determine if the calling object of this call is resolved to any abstract variables of interest.

Following the example in Figure 2.1, the abstract variable \((\text{calling object, LoaderManager, } mLoaders.get())\) used in defining the predicate can be paired with the method call at line 11, as the latter can be resolved to \((\text{calling object, LoaderManager, } mLoaders.put())\). Let us suppose \text{initLoader} has been invoked twice along the path: in the first API call, at line 11, we invoke \text{put} on the calling object’s field \(mLoaders\), and we then know in the second invocation of \text{initLoader}, at line 8, the call \text{get} on the same calling object will return a non-null value, and the branch at lines 9–13 will not be executed. The paired methods \text{get} and \text{put} helps to resolve the branch correlation in the two API calls.

### 2.2.4 Generating Summary Graphs

In the first three phases, we mark all the identified callback nodes, predicate nodes and update nodes on the ICFG of the API method. To add edges between the marked nodes and generate the summary graph, we traverse the ICFG and determine the reachability between the marked nodes in the ICFG with the goal that the final summary graph should keep the original control flow between the marked nodes.

Algorithm 3 takes as input the ICFG of the API method with the three types of marked nodes and generates a summary graph for the API method. The \text{worklist} at line 2 stores a pair of nodes, \(q\) and \(n\), where \(q\) is the last marked node seen and \(n\) is the node encountered during the traversal of the ICFG. This pair of nodes are always reachable from each other on the ICFG, as the two nodes initially are the same node (see line 3). When creating a new worklist element at line 12, the
successors of \( n \) are replaced, which are still able to reach \( q \). At lines 7 and 8, when we find that \( n \) is a marked node, an edge is added between \( n \) and \( q \).

A key function \( \text{Succ} \) at line 12 handles the challenges of interprocedural analysis. If \( n \) is the exit of the current method, the successors are located at the next statement of its call sites. If \( n \) is the call site, the successors can be found at the entry of its callees (only the ones containing marked nodes). A special case when a query reaches \( \text{sendMessage} \) in the ICFG, we find its successors in the inlined callbacks at the call site of \( \text{sendMessage} \) (see Section 4.1).

**ALGORITHM 3:** Generating the Summary Graph

```
Input : icfg = (N, E) ICFG of the API method
Output: SG = (Ns, Es) the summary graph of the API method
set SG to {} 
set worklist to {}
n0 = Entry(icfg); q = n0 // n0 is also the entry for summary
add (n0,q) to worklist
while worklist \( \neq \) {} do
    remove pair \((n,q)\) from worklist
    if \( n \) is the marked node or \( n \) is the exit of icfg then
        add edge \((q,n)\) to SG
        newq = n 
    end
    else newq = q;
    foreach \( s \in \text{Succ} (n, icfg) \) do add \((s,newq)\) to worklist;
end
return SG
```

2.3 Applying PCS

In this section, we show how we use generated PCSs to construct inter-callback ICFGs for Android apps and also to detect infeasible callback sequences.

As we mentioned, there are two major factors that determine the control flow of the app, external events, and Android API methods. Here, our focus is to sequence callbacks related to the Android API calls in the app. The functionalities of these callbacks include, but are not limited to, the Android lifecycles and component interactions.

To construct the app’s inter-callback ICFG, we start at some top-level method. It can be, for example, a handler for a GUI event. We first build the ICFG for this callback. We then traverse the ICFG, and when an Android API call is encountered, we add edges to connect the call site of
the API call to the entry of the summary, and also from the exit of the summary back to the call site. Next, we identify the implementation of the callbacks listed in the summary. To do so, we perform a pointer analysis on the app to identify the possible types of the calling object and the actual parameters of the API call. In case the parameter is an `Intent`, we resolve it to a set of possible component types defined in the app. Based on the types, we find the implementation of the callbacks invoked in the API methods at the call site.

To apply program analysis on the inter-callback ICFGs, we can use predicate nodes and update nodes in the summaries to prune the infeasible paths related to callback sequences. Detecting such infeasible paths can be useful to reduce the number of false positives in static analysis and help better estimate the coverage of testing.

To compute the infeasible paths on the inter-callback ICFGs of the apps, we implemented a demand-driven branch correlation algorithm [15]. We raise a query at each predicate node in the summaries. Then, we propagate this query backward along the paths of the CFG. The query can be resolved at the update node within the same PCS or in a different PCS. The modification of the algorithm here is that the query contains the abstract variables obtained from the predicate nodes, and we need to use the information from the update nodes to resolve the queries.

### 2.4 Experimental Results

The goals of our experiments are to show that 1) the PCSs generated are compact enough to be efficiently used by developers as well as static analysis and testing tools; 2) PCSs can be computed with practical precision and scalability; and 3) PCSs are useful for control flow analysis of Android apps.

#### 2.4.1 Experimental Setup

We implemented Lithium using Soot [61] for summarizing the Android API methods and for computing the apps’ inter-callback ICFGs. To summarize the Android API methods, we used as input the bytecode of the Android framework 5.1 implementation and applied Spark [35] to build
the call graphs for the API methods. To use Spark, we built a dummy main method which contains the calls to each of the Android API methods analyzed. To analyze apps, we used the .apk files as input and applied Dexpler [10] to convert them to the Soot Jimple representation.

As we mentioned, the Android framework has millions of lines of code and the call graph generated using Spark can be imprecise. To reduce the number of false positive callbacks due to call graph imprecisions, we used two heuristics: constraint the size of call chains and constraint the number of possible callers when generating call chains (explained in Section 2.2.1). The use of these heuristics can make our results unsound but reduces the number of false positives when the call graph blows up. For our experiments, we use 16 as the maximum length of the call chains and 5 randomly picked callers when traversing the call graph backward. For analyzing Android apps, we built the call graph for each callback in the app using Class Hierarchy Analysis (CHA).

To perform the experiments, we first evaluated close to 1000 Android apps from the Google Play Market and F-Droid [39]. Through analyzing the usage of the Android API methods in these apps, we identified 500 frequently invoked Android APIs and generated summaries for them. During summarization, we found a total of 193 PCSs that have at least one node and 127 PCSs have at least one callback. We selected 14 random Android apps from the F-Droid repository and constructed the apps’ inter-callback ICFGs using the generated summaries. We also generated dynamic traces through manual and random testing (Monkey [29]) to determine whether the paths in the apps’ inter-callback ICFGs can be found in real execution traces.

All of our experiments were run using a virtual machine (VM) with 4 cores and 40GB of memory. The VM runs on a machine with 16 cores of Quad Core AMD Operton 6204. We use a 64-bit JVM with a maximum heap size of 15GB. We provide the detailed experimental results in the next sections.

2.4.2 Compactness of the Summaries

For all the 500 Android API methods analyzed, we counted the number of nodes in their ICFGs as well as the number of nodes in their PCSs. By comparing the two, we found that the reduction
of PCSs over the ICFGs was on average 99%, and the maximum and minimum reductions found were 100% and 78% respectively.

We sorted the size of the ICFGs, PCSs, callback nodes, predication nodes and update nodes for the 500 API methods analyzed, and report the minimum, average and maximum number of nodes for each type in rows min, avg and max, under ICFG, PCS, Callback, P-Node and U-Node respectively, shown in Table 2.2. The results show that the size of the PCSs ranges from 0 to 20772 nodes with an average of 330 nodes.

Table 2.2: Size of ICFGs versus Size of Summary Graphs

<table>
<thead>
<tr>
<th></th>
<th>ICFG</th>
<th>PCS</th>
<th>Callbacks</th>
<th>P-Node</th>
<th>U-Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>avg</td>
<td>58604</td>
<td>330</td>
<td>55</td>
<td>154</td>
<td>120</td>
</tr>
<tr>
<td>max</td>
<td>208683</td>
<td>20772</td>
<td>5820</td>
<td>14794</td>
<td>3807</td>
</tr>
</tbody>
</table>

2.4.3 Correctness of the Summaries

In this section, we report our studies on the correctness of the PCSs generated by Lithium. In the first study, we compare the PCSs generated from API methods with ICFGs with 1000 nodes or less against a ground truth generated from manual analysis on the source code of the Android framework. From 310 methods that meet this criterion, 23 PCSs have at least one node. Table 2.3 reports a comparison on a total number of nodes found in the 23 PCSs and their ground truths. Under Match, we report the number of nodes that are in both the ground truths and the PCSs. Under Miss, we show the total number of nodes that are in the ground truth but missed by our PCS (false negatives). Under Additional, we list the number of nodes reported in the PCS but not present in the ground truths (false positives). The data show that the precision of the tool is 97% (just 4 false positives), and the recall is 85%. We found 4 callback nodes that were false positives because their object receivers were created in the framework which implies that the objects cannot be passed from any app (internal objects). For the false negatives, we found the main reason to be our heuristics on restricting the size of call chains and the number of callers at each call site to reduce the number of false positives. We found 7 API methods with false negative nodes. The
resulting 280 PCSs were proved to not have any node. PCS without nodes can be used by more conservative application call graph analyses such as *Averroes* [4] to reduce the number of edges introduced by possible callbacks.

<table>
<thead>
<tr>
<th></th>
<th>Match</th>
<th>Missed</th>
<th>Additional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callback</td>
<td>36</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>P-Node</td>
<td>95</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>U-Node</td>
<td>47</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.3: Comparison to Manually Identified Ground Truths

In our second study, we focused on analyzing callback nodes in PCSs with ICFGs with more than 1000 nodes. We took a sample of 300 callbacks nodes from 34 different PCSs to verify their correctness. The precision for this sample was 61%. The main reason for false positives we found was the imprecise call graphs obtained from the API methods. When we traverse the ICFG of the API method to obtain the call chains for a callback call site, we can encounter a large ICFG that contains a considerable number of virtual functions. The second source of imprecisions involves the object receivers being internal to the framework (the object is created in the framework and it cannot be an object from the app). For example, the PCS for the method `android.app.Dialog.show` calls callbacks for internal widgets used in windows for dialogs (these objects are internal to the framework). As we mentioned, as future work we consider implementing an analysis to detect all internal objects which can help to improve the precision of the PCSs.

### 2.4.4 Scalability of Generating the Summaries

In Figure 2.4a, we show the time used to build the PCSs for all the 500 API methods in ascending order. For 372 methods, the tool consumed less than 10 seconds with an overall average of 34.5 seconds. One of the aspects that contributes to the scalability of our tool was the demand-driven analysis that identified the call chains of the callbacks and discarded irrelevant methods for the expensive later phases.

In Figure 2.4b, we plot the performance against the size of the summary graphs. We inspected the API methods that consumed more than 1000 seconds and found that the backward symbolic
substitution analysis (used to resolve abstract variables in predicate nodes and identify update nodes) took more time in these methods because their call chains are longer than the rest of the methods. This behavior can increase the number of paths the analysis has to resolve, therefore, consuming more time.

Table 2.4: Constructing Apps’ inter-callback ICFGs Using PCSs

<table>
<thead>
<tr>
<th>App</th>
<th>Callbacks</th>
<th>API Calls</th>
<th>inter-callback ICFGs</th>
<th>Longest Path</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.blippex.app</td>
<td>21</td>
<td>4</td>
<td>1 1 2</td>
<td>2</td>
<td>0.2</td>
</tr>
<tr>
<td>net.sourceforge.andsys</td>
<td>22</td>
<td>12</td>
<td>1 6 15</td>
<td>9</td>
<td>0.1</td>
</tr>
<tr>
<td>com.darknessmap</td>
<td>25</td>
<td>11</td>
<td>1 2 5</td>
<td>4</td>
<td>0.1</td>
</tr>
<tr>
<td>de.onyxbits.remotekeyboard</td>
<td>41</td>
<td>29</td>
<td>1 2 4</td>
<td>5</td>
<td>0.2</td>
</tr>
<tr>
<td>com.example.android.contactslist</td>
<td>44</td>
<td>11</td>
<td>1 8 23</td>
<td>5</td>
<td>0.2</td>
</tr>
<tr>
<td>info.staticfree.SuperGenPass</td>
<td>50</td>
<td>20</td>
<td>2 4 7</td>
<td>5</td>
<td>0.0</td>
</tr>
<tr>
<td>com.markuspage.android.atimetracker</td>
<td>65</td>
<td>56</td>
<td>2 4 7</td>
<td>5</td>
<td>0.2</td>
</tr>
<tr>
<td>aarddict.android</td>
<td>66</td>
<td>28</td>
<td>1 3 10</td>
<td>13</td>
<td>0.3</td>
</tr>
<tr>
<td>de.ub0r.android.websms</td>
<td>73</td>
<td>89</td>
<td>0 3 8</td>
<td>6</td>
<td>0.3</td>
</tr>
<tr>
<td>com.google.zxing.client.android</td>
<td>83</td>
<td>14</td>
<td>1 5 34</td>
<td>84</td>
<td>0.2</td>
</tr>
<tr>
<td>a2dp.Vol</td>
<td>173</td>
<td>121</td>
<td>1 4 15</td>
<td>15</td>
<td>0.6</td>
</tr>
<tr>
<td>org.connectbot</td>
<td>176</td>
<td>108</td>
<td>1 6 27</td>
<td>43</td>
<td>1.1</td>
</tr>
<tr>
<td>org.openintents.filemanager</td>
<td>180</td>
<td>38</td>
<td>1 6 23</td>
<td>44</td>
<td>0.6</td>
</tr>
<tr>
<td>com.evancharlton.mileage</td>
<td>241</td>
<td>80</td>
<td>1 2 6</td>
<td>5</td>
<td>0.4</td>
</tr>
</tbody>
</table>

2.4.5 Constructing Apps’ inter-callback CFGs with PCSs

In this section, we report our results on control flow analysis of Android apps using PCSs. In Table 2.4, under App we present the name of the 14 apps studied. Under Callback, we report
a total number of callbacks implemented in the apps. Under *API calls*, we show the number of Android API calls that were connected to a PCS (using the PCSs generated for 500 API methods). We quantify the number of edges from a callback node in a PCS to the entry point of a callback method under *Apps’ inter-callback ICFGs*. Since we built partial inter-callback ICFGs for each top level method, we report the average, minimum and maximum number of edges for the partial graphs. We traverse each inter-callback ICFG to find the longest path in terms of callbacks, shown under *Longest Path*. Finally, we show the time used (in seconds) to build the inter-callback ICFGs under *Time (s)*. We do not include the time Soot took to build ICFGs for each top level method. The construction of the apps’ inter-callback CFGs took less than a second for all apps and 0.3 seconds on average. The results show that by modeling API calls using the summaries, we are able to connect callbacks to the control flow graphs with low overhead. The longest callback sequence found was of 44 callbacks for the app *org.openintents.filemanager*. Note that we used the summaries for the 500 most frequently used Android API method, and not all the API calls were modeled. We expect that the larger inter-callback ICFGs and the longer callback sequences would be generated if we plug in more summaries.

**Table 2.5: Compare Dynamic Traces and Static Paths**

<table>
<thead>
<tr>
<th>App</th>
<th>Traces covered-c</th>
<th>Traces total</th>
<th>Traces covered-p</th>
<th>Infeasible</th>
<th>Paths in traces</th>
<th>Paths not in traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.blippex</td>
<td>94%</td>
<td>12</td>
<td>12</td>
<td>0</td>
<td>77.8%</td>
<td>22.2%</td>
</tr>
<tr>
<td>net.sourceforge.andsys</td>
<td>81%</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>47%</td>
<td>53%</td>
</tr>
<tr>
<td>com.darknessmap</td>
<td>64%</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>63%</td>
<td>37%</td>
</tr>
<tr>
<td>com.example.android.contactslist</td>
<td>80%</td>
<td>6</td>
<td>6</td>
<td>12.5%</td>
<td>67.5%</td>
<td>20%</td>
</tr>
<tr>
<td>de.onyxbits.remotekeyboard</td>
<td>66%</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>56.2%</td>
<td>43.8%</td>
</tr>
<tr>
<td>info.staticfree.SuperGenPass</td>
<td>64%</td>
<td>6</td>
<td>6</td>
<td>20.8%</td>
<td>58.4%</td>
<td>20.8%</td>
</tr>
<tr>
<td>com.markupage.android.atimetracker</td>
<td>47%</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>41%</td>
<td>59%</td>
</tr>
<tr>
<td>aarddict.android</td>
<td>65%</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>3%</td>
<td>97%</td>
</tr>
<tr>
<td>de.ub0r.android.websms</td>
<td>43%</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>21.5%</td>
<td>78.5%</td>
</tr>
<tr>
<td>com.google.zxing.client.android</td>
<td>57%</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0.3%</td>
<td>99.7%</td>
</tr>
<tr>
<td>org.openintents.filemanager</td>
<td>35%</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>4.2%</td>
<td>95.8%</td>
</tr>
<tr>
<td>org.connectbot</td>
<td>41%</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>2%</td>
<td>98%</td>
</tr>
<tr>
<td>a2dp.Vol</td>
<td>47%</td>
<td>10</td>
<td>9</td>
<td>0</td>
<td>0.4%</td>
<td>99.6%</td>
</tr>
<tr>
<td>com.evancharlton.mileage</td>
<td>40%</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>33.8%</td>
<td>66.2%</td>
</tr>
</tbody>
</table>
In Table 2.5, we show the comparison of the traces generated from testing the apps (see Section 2.4.1) and the paths (the callback sequences) generated from the inter-callback ICFGs. Column Traces reports the data obtained from analyzing the traces. Under covered-c, we report the percentage of the callbacks covered in the traces. Under total, we report the total number of runs done for generating traces. Under covered-p, we report how many of these traces contain the callback sequences obtained from the apps’ inter-callback ICFGs. We perform this comparison by verifying whether the paths of callback sequences are subsequences of the dynamic traces. Once we integrate our analysis with a GUI model, we would be able to compare the program paths against the complete traces. Our results show that on average, we covered 49% of the callbacks during testing. We partially covered 96 out of 97 traces from the 14 apps. For app such as a2dp.Vol, we found that the tests did not reach some of the API calls we modeled. Column Paths report the analysis of paths. Under infeasible, in traces and uncovered, we report the number of infeasible paths of callbacks detected using the predicate and update nodes; the number of paths of callback sequences that were found in the dynamic traces; and the number of paths of callback sequences that were not found in the traces. Among 6 apps, we were able to confirm that more than 50% of the paths are either infeasible or covered by the traces. Therefore, even with small number of PCSs used in the app analysis, we are able to generate valid paths in some apps. Regarding the apps with low number of paths found in traces, we observed that the traces for these apps had low callback coverage.

2.5 Discussion

The main source of imprecisions for generating PCSs came from the call graphs generated for API methods. As we mentioned before, they tend to blow up given the complexity of the Android’s framework source code. We used the maximum length of call chains and the maximum number of callers as thresholds to restrict the scope of the analysis to reduce the false positives. This sampling heuristics may lead to incomplete numbers of callback, predicate and update nodes. For future work, we plan to use more precise call graph and points-to analysis algorithms to reduce the
number of false positives. In addition, our current analysis does not handle callbacks with object receivers passed from different API methods to the framework. We plan to study a solution to resolve such objects during client analysis.

2.6 Summary

This chapter presents a static analysis technique to construct control flow graphs of Android apps related to callback sequences. The novelty of the work is a specification technique named predicate callback summaries (PCS) designed to model the control flow of callbacks implemented in the API method. We presented the definition, computation, and applications of the specification in our framework Lithium. Our experiments reported that using the PCSs generated, we can connect up to 44 callbacks in a path, which previously was only done manually. We are also able to prune infeasible callback sequences, which can improve the precision of static analysis and testing.
CHAPTER 3. SPECIFYING CALLBACK CONTROL FLOW OF ANDROID APPS USING FINITE AUTOMATA

In this chapter we present a novel program representation, namely Callback Control Flow Automata (CCFA), to specify control flow of callbacks for event-driven and framework-based applications. The design of CCFA is based on the Extended Finite State Machine (EFSM) [51] model, which extends the Finite State Machine (FSM) by labeling transitions using information such as guards. Using the CCFAs, we aim to specify four types of callback control flow existing in an app: 1) a callback $B$ is invoked synchronously after another callback $A$, 2) $B$ is invoked asynchronously after $A$, meaning $B$ is put in the event queue of the Android system after $A$ is invoked, 3) during an execution of $A$, $B$ is invoked synchronously by an API call, and 4) during an execution of $A$, $B$ is invoked asynchronously by an API call.

In our CCFAs, a state indicates whether the execution path enters or exits a callback. The transition from one state to another represents the transfer of control flow between callbacks. The transitions are labeled with guards, the conditions under which a transition is enabled. Such a condition can be an external event (e.g. click the button) or an API call. The input alphabets of the CCFA are the names of callbacks implemented in the app. Specifically, we use $\text{callbackname}_{\text{entry}}$ and $\text{callbackname}_{\text{exit}}$ as input symbols so we can specify the case where one callback is invoked during the execution of the other (see types 3 and 4 control flow discussed in the previous paragraph). Using this approach, the language recognized by a CCFA is a set of paths of callbacks or possible sequences of callbacks that can be invoked at runtime for an app.

To compute CCFAs, we designed a static analysis algorithm that traverses Windows Transition Graph (WTG) [66] to obtain the control flow of callbacks related to the windows (activities, dialogs and menus) and GUI events, and also integrates Predicate Callback Summaries (PCSs) [49] for callback sequences invoked in the Android API methods. These API methods include the callbacks...
of the Android components’ lifecycles, which are previously modeled manually. Our approach is able to automatically handle them. We consider that CCFAs and our analysis are easily extensible to future control flow models for callbacks not covered by WTGs and PCSs, e.g., the handlers of camera events and GPS location updates. Callback control flow graphs are important for analyzing the global behaviors of the apps.

```java
class A extends Activity {
    Button b1;
    void onCreate(Bundle b) {
        b1 = (Button) findViewById(...);
        b1.setOnClickListener(new CList());
    }
    void onStart() {
        ...
        if (*) { ...
            L l = new L();
            LoaderManager lm = loaderManager();
            lm.initLoader(0, null, l);
        }
    }
    void onStop() { ...
}
}

class L implements LoaderCallbacks {
    Loader onCreateLoader(int i, Bundle b) {
        ...
    }
}

class CList extends OnClickListener {
    void onClick(View v) { ...
}
}
```

(a) Source code of a simple Android app

3.1 An Overview

Given the event-driven, framework-based architecture of Android apps, most of an app’s implementation is done by defining callbacks that are invoked by the Android framework. To perform inter-callback static analysis or testing the apps beyond unit level, it is important to discover and represent the potential orders of the callbacks. In this section, we use an example to provide an overview of what is a CCFA and also the approach of computing the CCFA.

![CCFA for the Android app](image)

(b) CCFA for the Android app

![WTG for the Android app](image)

(c) WTG for the Android app

![PCS for initLoader](image)

(d) PCS for initLoader

Figure 3.1: Source code, CCFA, WTG and PCS of an example app
3.1.1 What is a CCFA

In Figure 3.1a, we show a simple Android app that contains one activity class called A. This activity implements the callbacks `onCreate` (line 3), `onStart` (line 7) and `onStop` (line 15). In the callback `onCreate` at line 5, the app sets an object of type `CList` as the event listener for the click event of the button `b1`. It also invokes the API call `lm.initLoader` at line 12 to load the data, and this API call uses the class L (lines 17-22) to create a loader.

Figure 3.1b shows the CCFA we constructed for Figure 3.1a. The representation is designed based on the Extended Finite State Machine (EFSM), and each transition is labeled with a pair \((x, g)\) where \(x\) is an input symbol and \(g\) is a guard. The input symbol \(x\) can represent a callback’s entry point (see \(A.onCreate_{entry}\) on the transition from state \(q_1\) to \(q_2\)), a callback’s exit point (see \(A.onCreate_{exit}\) on the transition from state \(q_2\) to \(q_3\)), or an empty input represented by \(\epsilon\) (see transition from state \(q_4\) to state \(q_6\)). Similar to the transitions in an EFSM, the inputs of CCFAs can be parametrized using variables. As examples, in Figure 3.1b, on the transition from state \(q_6\) to state \(q_7\), the input \(L.onCreateLoader_{entry}\) has a parameter \(cs\), and it is used in the guard, the boolean expression \(cs = lm.initLoader(0, null, 1)\), to specify that when the call site `lm.initLoader` at line 12 is encountered, the control flow transfers to the entry of the callback `L.onCreateLoader`. In case any other call site is encountered, the transition cannot be enabled. Similarly, the transition from state \(q_5\) to state \(q_{10}\) indicates that the parameter of input `CList.onClick_{entry}` is `evt`, and when the event of `click_b1` is triggered, the callback `CList.onClick` will be invoked. For the transitions where the control flow transfer always happens, we use the `true` guard.

This representation specifies both synchronous and asynchronous invocations of callbacks, and we are able to recognize them based on the transitions’ parameters and guards. Sequential invocation of callbacks is specified using the guard `true` or the guard related to the `cs` parameter. As an example, the transition from state \(q_3\) to state \(q_4\) in Figure 3.1b indicates that the execution of \(A.onStart\) immediately follows \(A.onCreate\). CCFAs can also specify synchronous callback invocations from an API call when its calling callback has not finished its execution (see transitions between states \(q_4, q_6\) and \(q_7\) in Figure 3.1b). Callbacks that are parametrized with external events are ex-
3.1.2 How to Compute a CCFA

In Figure 3.2, we show that our approach of computing CCFAs takes a Windows Transition Graph (WTG) [66], the interprocedural control flow graphs (ICFGs) of callbacks in the app, and the Predicate Callback Summaries (PCSs) of the Android framework [49] as inputs. The WTG contains pre-computed callback control flow related to windows (Activities, Menus or Dialogs) and GUI events, and the PCS specifies the pre-computed callback control flow for the Android framework methods.

Performing on the three inputs, our approach consists of three major analyses: WTG Analysis, Callback Analysis and PCS Analysis. As shown in Figure 3.2, in the first step, the WTG Analysis traverses the WTG to obtain callback sequences located on the edges of WTG. For each callback, the WTG Analysis invokes the Callback Analysis, an interprocedural analysis on the ICFG of
the callback, aiming to compute a CCFA that specifies the callback control flow implemented in
the callback. The key role of a Callback Analysis is to integrate the callback control flow related
to the API methods invoked in the callback. For each API call, the Callback Analysis finds the
corresponding PCS and calls the PCS Analysis. The PCS Analysis traverses the PCS and generates
a CCFA that represents all the sequences of callbacks executed in the API method. The output
of the PCS Analysis is a CCFA for the given API call, which is used by the Callback Analysis.
Each Callback Analysis generates a CCFA that represents all the sequences of callbacks generated
from API calls done from the callback’s ICFG. This CCFA is then composed in the WTG analysis.
Once the WTG Analysis traverses all the edges in WTG, it returns the final CCFA for the app.

Using this approach, we correctly integrated the callback control flow from the three sources. In
Figure 3.1b, the transitions between states $q_1, q_2, q_3$ and $q_4$ represent the sequential execution of the
callbacks A.onCreate and A.onStart obtained from the WTG in Figure 3.1c. The Callback Analysis
is flow and context-sensitive analysis and able to order the API methods invoked on the paths and
also to distinguish the paths that invoke the API method and the paths that do not. At line 7 in
Figure 3.1a, the callback A.onStart has two paths and only one of the paths invokes the API call
lm.initLoader. Correspondingly, in Figure 3.1b, the transition from state $q_4$ to state $q_5$ represents
the path that does not invoke the API call, while the transitions $q_4 \rightarrow q_6 \rightarrow q_7 \rightarrow q_8 \rightarrow q_5$ include
the callback L.onCreateLoader invoked in the API method. In addition, Figure 3.1b indicates that
the API call is invoked during the execution of A.onStart, as the transitions related to the API
methods are located after A.onStart_entry and before A.onStart_exit. We also correctly retrieve the
execution order of callbacks invoked within the Android API method from the PCS. The transitions
between states $q_6, q_7$ and $q_8$ in Figure 3.1b are generated by the PCS Analysis on the PCS shown
in Figure 3.1d.

3.2 Callback Control Flow Automata

In this section, we define our representation, Callback Control Flow Automata (CCFA) and
clarify its capabilities of modeling actual runtime behaviors of Android apps.
3.2.1 Definition

The CCFA is a representation based on the computation model of Extended Finite State Machines (EFSMs) [19].

**Definition 1 (CCFA).** A CCFA over a finite set of input symbols $X$ and input parameters $R$ is a 4-tuple $(S, s_0, T, F)$, where

- $S$ is a finite set of states,
- $s_0 \in S$ is the initial state,
- $T$ is the set of transitions, and each transition $t \in T$ is a tuple $(s_1, x, g, s_2)$, where $s_1 \in S$ and $s_2 \in S$ are the two states, $x \in X$ is the input symbol, and $g$ is the guard, a predicate over the input parameters, and
- $F \subseteq S$ are a set of final states.

**Input Symbols.** The CCFA specifies the control flow transfer between callbacks. The input symbols are the names of the callbacks. Given a finite set of callback names of an app, $C$, we use the strings $\{x_{\text{entry}} | x \in C\}$ to denote the entry points of the callbacks (the entry node of the ICFG of the callback), and $\{x_{\text{exit}} | x \in C\}$ for the exit points (the exit node of the ICFG). The input symbols are defined as: $X = \{x_{\text{entry}} \cup x_{\text{exit}} | x \in C\}$. As an example, if an app has one activity component, called $A$, that defines the two callbacks onCreate and onStart, we have the input symbol $X = \{A.onCreate_{\text{entry}}, A.onCreate_{\text{exit}}, A.onStart_{\text{entry}}, A.onStart_{\text{exit}}\}$. We use the entry and exit points of callbacks to be able to handle the following control flow relations: 1) one callback is invoked after the other callback, and 2) one callback is invoked during the execution of another callback.

**Input Parameters and Guards.** Similar to EFSMs, input symbols of CCFAs can be parametrized. The input parameters are used by guards to specify the conditions of a transition. That is, the guards are predicates over input parameters that decide whether a transition can be fired. Each
input on the transition can take input parameters of the following three types: \textit{evt}, the external events, \textit{cs}, the API call sites, and \textit{msg}, the asynchronous messages.

\textit{evt}: In event-driven applications, an event listener is a callback that is executed to respond to an external event from the environment. In CCFAs, transitions to entry points of event listeners/callbacks are parametrized by the variable \textit{evt}. This variable carries information about the specific event that triggers the invocation of the callback. For example, the transition from states $q_5$ to $q_{10}$ in Figure 3.1b is labeled with the input \textit{Clist.onClick\_entry}, the input parameter for this input is \textit{evt}, and the guard that uses the input parameter is $\text{evt} = \text{click}_{b1}$, indicating that the event \text{click}_{b1} triggers the transition. When computing CCFAs, we obtain the list of events and their orders from the WTG. Among all the external events, a WTG focuses on modeling GUI events and the events that affect the state of windows. Specifically, a GUI event is a specific interaction of a user with a widget or view defined in an app (e.g., user clicks on a button). External events that affect the state of windows include hardware buttons \textit{back button} or \textit{power button}, and the actions such as \textit{rotation of the phone}.

\textit{cs}: The second factor that impacts the control flow of the callbacks is the Android API calls. To label the transition in the CCFA, we use the input parameter \textit{cs} with the input symbol $x_{\text{entry}}(cs)$ and the guard $cs = y.api$ to specify that at the API call site $y.api$, the control flow is transferred to the entry of the callback $x$, for $x$ is the first callback to be executed in the \textit{api} method. As a concrete example, in Figure 3.1b, the transition from states $q_6$ to $q_7$ is labeled with the input symbol \textit{L.onCreateLoader\_entry} and its parameter \textit{cs}, and the guard $cs = \text{lm.initLoader}(0, \text{null}, 1)$. The transition indicates that \textit{L.onCreateLoader} is the first callback executed in the API call \textit{initLoader}.

\textit{msg}: There are two special Android API methods, \texttt{Handler.sendMessage} and \texttt{Handler.post}, that provide an asynchronous message-passing mechanism for an app. When the two APIs are invoked in the app, an instance of Handler will post messages and \texttt{runnable objects} respectively to the event queue. Correspondingly, when the message is dequeued, the framework execute callbacks \texttt{Handler\_handleMessage} and \texttt{Runnable\_run} respectively. In CCFAs, the transitions to the entry points of callbacks \texttt{Handler\_handleMessage} (or the callback \texttt{Runnable\_run}) are parametrized using the input
```java
1 class A extends Activity {
2     MyH h = new MyH();
3     ...
4     void onPause() {
5         ...
6         h.sendMessage(m);
7     }
8 }
9 class MyH implements Handler {
10     void handleMessage(
11         Message msg) { ... }
12 }
```

(a) User-defined Handler called in A.onPause

(b) CCFA for A.onPause

Figure 3.3: Guard for sendMessage

parameter `msg`. The variable `msg` carries information about the specific `Handler.sendMessage` (or `Handler.post`) call site that triggers the asynchronous execution of the callback. As an example, Figure 3.3a shows an app that defines its own handler (`MyH` in lines 9-12) and this handler is used by activity `A` in the callback `onPause`. The API call to `Handler.sendMessage` at line 6 is represented by the transition from state `q8` to state `q9` in Figure 3.3b. The transition is labeled with the callback name `MyH.handleMessage_entry`, the input parameter `msg`, and the guard `msg = h.sendMessage(m)`.

### 3.2.2 The Scope of CCFAs

To the best of our knowledge, the Android framework supports four types of control flow between callbacks: 1) a callback `B` is invoked synchronously after another callback `A`, 2) `B` is invoked asynchronously after `A`, meaning `B` is put in the event queue of the Android system and invoked eventually after `A` is invoked, 3) during an execution of `A`, `B` is invoked synchronously by an API call, and 4) during an execution of `A`, `B` is invoked asynchronously by an API call. Our CCFA is able to specify all the four cases. For the asynchronous invocation, we support the guards that use `evt` and `msg` as input parameters. Although the actual generation of CCFAs used WTGs as inputs, we believe that our representation is sufficient to specify the guards with events beyond GUI and Windows, e.g., sensor’s events. Meanwhile, for other special Android constructs such as
Fragments, we may need to include special mechanisms for asynchronous invocations, including more types of input parameters.

It should also be noted that the paths in CCFAs represent the orders of invocations of callbacks in an app. In the presence of asynchronous invocations, the order of invocations are not always the order of executions; when the system invokes a call asynchronously, the callback is put in the queue rather than directly executed.

### 3.3 Computing a CCFA

In this section, we introduce the algorithms to construct the CCFAs from the source code or binary of an app. The algorithms follow the process presented in Section 3.1 in Figure 3.2 and consist of three main procedures that call each other to generate a final CCFA. Specifically, Algorithm 4 represents WTG Analysis in Figure 3.2, Algorithm 5 represents Callback Analysis, and Algorithm 6 represents PCS Analysis. Since our algorithms heavily use the WTGs and PCSs representations, we formally introduce them as follows.

A Windows Transition Graph (WTG) specifies the possible GUI window sequences with their corresponding events and handlers [66]. Let \( \text{Win} \) be the set of all windows (activities, menus or dialogs) defined in an app and \( \text{C} \) be the set of callbacks implemented in an app. A WTG is a directed graph \( G = (\text{Win}, \text{E}, \lambda, \delta, \sigma) \) with nodes \( w \in \text{Win} \) and edges \( e \in \text{E} \subseteq \text{Win} \times \text{Win} \). The initial node in the graph is a LAUNCHER node, and it is connected to the main activity declared in the file AndroidManifest.xml. Edges are labeled with information obtained via functions \( \lambda, \delta \) and \( \sigma \). The function \( \lambda : \text{E} \to \text{Event} \) returns the event that triggered the transition, whereas the function \( \delta : \text{E} \to (\text{push, pop} \times \text{Win})^* \) returns the window stack operations. The most relevant to this work is the function \( \sigma : \text{E} \to ((\text{Win} \cup \text{View}) \times \text{C})^* \) that returns the sequence of callbacks potentially executed during the transition. Figure 3.1c presents an example of WTG.

A Predicate Callback Summary (PCS) summarizes the control flow of callbacks invoked in the Android API methods [49]. A PCS is a directed graph \( G = (N_c \cup N_p \cup N_u, \text{E}) \) where \( N_c \) is the set of callback nodes, \( N_p \) is the set of predicate nodes and \( N_u \) is the set of update nodes that modify the
variables used in the predicate nodes. \( E \) is the set of edges between any two nodes. The important nodes to our work are the callback nodes. A callback node is specified using a call signature and the object receiver of the callback. Figure 3.1d presents a simple example of PCS, where the callback `onCreateLoader` is invoked and the object receiver represents the third parameter of the API call.

**Algorithm 4:** Compute a CCFA for an app \( P \)

\[
\begin{array}{l}
\textbf{input}: \ WTG \ \text{(model for windows and GUI)}, \ P \ \text{(App code)}, \ Z \ \text{(PCSs)} \\
\textbf{output}: \ CCFA \ \text{(Callback Control Flow Automata)} \\
\text{init} \leftarrow \text{getState(entry(WTG))}; \ \text{Seen} \leftarrow \{(\text{entry(WTG)}, \text{init})\}; \\
\text{Worklist} \leftarrow \{(\text{entry(WTG)}, \text{init})\}; \ CCFA \leftarrow (\text{init}, \{\}, \{\}); \\
\text{while} \ \text{Worklist} \neq \emptyset \ \text{do} \\
\quad (\text{node}, s_0) \leftarrow \text{remove(Worklist)}; \\
\quad s_1 \leftarrow \text{getState(node)}; \\
\quad \text{foreach} \ \text{edge} \in \text{succ(node, WTG)} \ \text{do} \\
\quad \quad g \leftarrow \text{createEvtGuard(edge)}; \\
\quad \quad \text{while} \ c = \text{getNextCallback(edge)} \ \text{do} \\
\quad \quad \quad \text{if} \ c \ \text{is last} \ s_2 \leftarrow \text{getState(edge.tgt)}; \\
\quad \quad \quad \text{else} \ s_2 \leftarrow \text{getState(null)}; \\
\quad \quad \quad \text{CCFA} = \text{extend(CCFA, c, s_1, s_2, s_0, g)}; \\
\quad \quad \quad g \leftarrow \text{true}; \ s_1 \leftarrow \text{getState(null)}; \ s_0 \leftarrow s_2; \\
\quad \quad \text{end} \\
\quad \quad \text{if} \ (\text{edge.tgt, s_0}) \notin \text{Seen} \ \text{then} \\
\quad \quad \quad \text{add(edge.tgt, s_0) to Seen and Worklist} \\
\quad \quad \text{end} \\
\quad \quad \text{end} \\
\quad \text{end} \\
\quad \text{if} \ \text{node is exit then} \ CCFA.\text{finalStates} = CCFA.\text{finalStates} \cup \{s_0\}; \\
\text{end} \\
\text{procedure extend(CCFA, c, s_1, s_2, s_0, g)} \{ \\
\quad \text{add transition (s_0, c.entry, g, s_1) to CCFA;} \\
\quad F_c \leftarrow \text{analyzeCallback(c, s_1)}; \\
\quad \text{if} \ F_c \ \text{is not empty then} \\
\quad \quad \text{add all transitions from } F_c \ \text{to CCFA;} \\
\quad \quad \text{foreach } s_f \in F_c.\text{finalStates} \ \text{do} \\
\quad \quad \quad \text{add transition (s_f, c.exit, true, s_2) to CCFA;} \\
\quad \quad \text{end} \\
\quad \text{end} \\
\quad \text{else} \ \text{add transition (s_1, c.exit, true, s_2) to CCFA;} \\
\quad \text{return CCFA;} \\
\}
\]

Shown in Algorithm 4, our analysis takes the window transition graph, \( WTG \), the source or binary code of an app, \( P \), and a set of pre-generated PCSs for Android API methods, \( Z \), as inputs. At a high level, the algorithm uses a worklist to traverse the WTG at lines 3 to 19, obtains all the
callbacks in the WTG’s edges using the function `getNextCallback` at line 8, and extends the CCFA at line 11 with the states and transitions constructed from the callbacks.

At lines 1–2, `init` is the initial state of the CCFA. `Seen` is the flag that marks whether a node in the WTG has been processed. Each element in `Worklist` is a pair of `node`, a node in the WTG currently under processing, and `s0`, the current state in the CCFA under construction to which the new generated transition should connect. Finally, `CCFA` stores the representation building in progress, specified with 3 elements: `init`, the initial state as well as a set of transitions and a set of final states (both of them are set to empty sets initially at line 2).

At line 5, the algorithm creates the state `s1` for the WTG node using `getState`, and at lines 9 and 10, we create the state `s2`. The `getState` function returns a unique state for each node in the WTG; when invoked with the `null` parameter, it returns a new state for the next callback in the same edge of the WTG (see line 10). At line 7, the algorithm takes the event from the WTG edge and creates a guard for this transition using `createEvtGuard`. At line 11, the function `Extend` takes the CCFA under construction, the states `s0`, `s1`, and `s2`, the callback `c`, and the guard `g` as inputs, and creates two transitions, one from `s0` to `s1` (see line 21), and the other from `s1` and `s2` (see line 29). It also analyzes the callback `c`, resolves any API calls in the callback, and generates a CCFA `Fc` for the callback `c` (see line 22). `Fc` takes `s1` as its initial state. The details of `analyzeCallback` are given in Algorithm 5. At lines 14–16, the target node of this edge, `edge tgt`, is added to the worklist if it has not been processed before. At line 18, the exit node of the WTG is encountered, and the last state that has been seen, `s0`, is marked as a final state.

Algorithm 5 presents `analyzeCallback`, an interprocedural, flow-sensitive analysis on the ICFG of a given callback. At lines 1–2, the algorithm constructs the ICFG for the given callback `c` and gets the entry node `n0`. It then performs the initializations for the set of data structures `Worklist`, `Seen` and `F`, similar to Algorithm 1.

The worklist algorithm runs from lines 3 to 27. At line 4, `n` is an ICFG’s node, and `s0` is the state last added to `F` to which we aim to connect. At line 5, `n` is detected as a call to the API method `sendMessage` or `post` of the `Handler` class. We get the corresponding callbacks
Handler.handMessage or Runnable.run using getCallback, and create the guard \( g \) using function createMsgGuard at line 6. Specifically, the function getCallback finds the corresponding callback by getting the type of the object receiver for sendMessage or the first parameter’s type for post calls using the pointer analysis. For example, in Figure 3.3 the transition from state \( q_8 \) to \( q_9 \) is labeled with \( MyH.handleMessage \) entry because the object receiver of the call at line 6 in Figure 3.3a has the type \( MyH \). Finally, at line 8, the extend function from Algorithm 4 is used to add the transitions for the callback \( c \).

In the second case, at line 10, \( n \) is detected as an API call site. At line 11, we use the function analyzePCS (details in Algorithm 6) to generate a CCFA for the API call and store it in \( F_{PCS} \). If \( F_{PCS} \) is not empty, we add all the transitions of \( F_{PCS} \) to the current CCFA \( F \) (line 13). We also create epsilon transitions from \( s_0 \) to the initial state of \( F_{PCS} \) and from the final states of \( F_{PCS} \) to a newly created state \( s_0 \) at line 15. This new state becomes the last state for this path which is added to the worklist at line 24. To explain this part of the algorithm using an example, in Figure 3.1b, the \( F_{PCS} \) returned for the API call \( lm.initLoader \) includes transitions from \( q_6 \) to \( q_8 \). The process of concatenating \( F_{PCS} \) to the current CCFA adds epsilon transitions from state \( q_4 \) to \( q_6 \) and from \( q_8 \) to \( q_9 \).

At line 21, the current statement is determined to be the exit node of the ICFG, and we thus add \( s_0 \) to the final states in the CCFA. At lines 22–25, we continue fetching the successor nodes on the ICFG for further processing. Our current algorithm creates new states every time a callback is added, and in case of a loop, we iterate the loop once to ensure the termination of the analysis. In the future, we plan to integrate an abstraction similar to the approach presented by Shohan et al. [55] to handle loops.

In Algorithm 6, we show our approach of generating the CCFA for a given API call \( stmt \). At line 1, we find the PCS for the API call \( stmt \) and creates the guard based on its call site. At line 2, we applied a pointer analysis to resolve the types of the API call’s object receiver and parameters (in our implementation, we used the pointer analysis provided by Soot [61]). The algorithm uses a worklist to traverse the PCS at lines 6 to 18.
Algorithm 5: Function analyzeCallback used in Algorithm 4

input: $c$ (Callback), $s$ (initial state)
output: $F$ (CCFA for callback $c$ with initial state $s$)

$G \leftarrow \text{buildICFG}(P, c)$; $n_0 \leftarrow \text{entryNode}(G)$;
Worklist $\leftarrow \{(n_0, s)\}$; Seen $\leftarrow \{(n_0, s)\}$; $F \leftarrow (s, \{\}, \{\})$;
while Worklist $\neq \emptyset$ do
    $(n, s_0) \leftarrow \text{remove}(\text{Worklist})$;
    if $n$ is a call to sendMessage or post then
        $c \leftarrow \text{getCallback}(n)$; $g \leftarrow \text{createMsgGuard}(n)$;
        $s_1 \leftarrow \text{getState}(null)$; $s_2 \leftarrow \text{getState}(null)$;
        $F \leftarrow \text{extend}(F, c, s_1, s_2, s_0, g)$; $s_0 \leftarrow s_2$;
    end
    else if $n$ is an API call then
        $F_{PCS} \leftarrow \text{analyzePCS}(n)$;
        if $F_{PCS}$ is not empty then
            add all transitions from $F_{PCS}$ to $F$;
            add transition($s_0, \epsilon, \text{true}, F_{PCS}.\text{initialState}$) to $F$;
            $s_0 \leftarrow \text{getState}(null)$;
            foreach $s_f \in F_{PCS}.\text{finalStates}$ do
                add ($s_f, \epsilon, \text{true}, s_0$) to $F$;
            end
        end
    end
    else if $n$ is an exit node then $F.\text{finalStates} = F.\text{finalStates} \cup \{s_0\}$;
    foreach succ $\in \text{succs}(n, G)$ do
        if $(\text{succ, } s_0) \notin \text{Seen}$ then
            add $(\text{succ, } s_0)$ to Worklist and Seen
        end
    end
end
return $F$;
At lines 8–9, we create states \( s_1 \) and \( s_2 \) for a callback node encountered. At line 10, we set the guard for the current transition: if the previous state \( s_0 \) is the initial state \( \text{init} \), the guard is \( g_{cs} \); otherwise the guard is \( \text{true} \) (only the transitions to the entry point of the first callbacks in the API methods have a guard regarding the API call site). At lines 11-12, we use the function \( \text{findCallbacks} \) to obtain the possible implementations of callbacks of the current node \( n \). This function first gets the possible types of the callback using \( \text{refs} \) and the object receiver of \( n \). For example, if the object receiver in the callback is a parameter of the API call, this function finds all the possible types of the formal parameter at the API call. Then, it finds the callbacks implemented in these types using the callback node’s signature. In case the object receiver in the callback node in PCS is \( \text{unknown} \), the algorithm conservatively identifies all the callbacks implemented in the app that matches the given callback’s signature. At line 12, \( F \) is extended by integrating the callback \( c \). If the node is the exit node of the PCS (line 16), we add the last state seen \( s_0 \) as a final state of \( F \). The function \( \text{propagatePCS} \) finds the successors of the current node \( n \) and add them to the worklist.

**Algorithm 6:** Function \( \text{analysePCS} \) used in Algorithm 5

```plaintext
input : stmt (API call site)  
output: F (CCFA for the API call)  
PCS ← getPCS(Z, stmt); g_{cs} ← createAPIGuard(stmt);  
refs ← resolveReferences(stmt);  
n_0 ← entryNode(PCS); init ← getState(null);  
Worklist ← \{\( (n_0,\text{init}) \)\}; Seen ← \{\( (n_0,\text{init}) \)\};  
F ← (\( \text{init} \), \{\}, \{\});  
while Worklist \( \neq \emptyset \) do  
  \( (n, s_0) \) ← remove(Worklist);  
  if \( n \) is callback node and \( P \) implements it then  
    \( s_1 ← \text{getState(null)}; s_2 ← \text{getState(null)}; \)  
    if \( s_0 = \text{init} \) then \( g = g_{cs} \) else \( g ← \text{true} \);  
    foreach \( c ∈ \text{findCallbacks}(n, P, \text{refs}) \) do  
      \( F ← \text{extend}(F, c, s_1, s_2, s_0, g) \);  
    end  
    \( s_0 ← s_2 \);  
  end  
  else if \( n \) is exit node then  
    \( F.\text{finalStates} = F.\text{finalStates} \cup \{s_0\} \);  
    \( \text{propagatePCS}(n, PCS, s_0, \text{Worklist}, \text{Seen}) \);  
  end  
return \( F \);  
```
We extend the CCFA with 9 external events using models manually generated from the official documentation and testing. For most of the external events, the app registers an object that listens for different events. For example, the API method `requestLocationUpdates` from the class `LocationManager` registers an object of type `LocationListener`. The framework executes callbacks from this object when it reacts to the GPS location updates, and enables and disables the GPS antenna. In our analysis, we extend the callback analysis (Algorithm 2) to detect where the objects are registered and disabled, and we then insert the transitions of callbacks correspondingly based on the models.

### 3.4 Program Analysis using CCFAs

In this section, we show the two approaches for integrating CCFAs to perform inter-callback analysis. In the first approach, we extend an existing interprocedural, context-sensitive dataflow analysis [53] using CCFAs. The key idea is to query CCFAs for the successor callbacks whenever the analysis reaches the end of a callback or an API call site within the callback. In the second approach, we generate the main function by integrating all the callback sequences provided by CCFAs. This approach is similar to how FLOWDROID [8] integrates the lifecycle model. In both of the approaches, we traverse the paths of the callback invocations provided by the CCFAs without considering the potential impact of the event queue, and the guards regarding external events are treated as `true`.

#### 3.4.1 First Approach

In Algorithm 7, we present the inter-callback dataflow analysis extended from [53]. The modification is that we add the cases when the analysis reaches the end of a callback and when the analysis reaches an Android API call site. The inputs of the algorithm include `ICFGs`, the set of ICFGs of the callbacks, $\mathcal{L}$, the semi-lattice of dataflow facts, and $x_0$, the initial context. Let $N$ be the set of all nodes in the ICFGs. The output $S[N]$ reports the dataflow facts for all the nodes in the app.
Algorithm 7: Inter-callback Dataflow Analysis using CCFAs

**Input:** ICFCs (set of ICFCs for the callbacks in the app), \( L \) (semi-lattice), \( F \), (CCFA for the app), \( x_0 \) (an initial context)

**Output:** \( S[N] \) of \( L \)

**Foreach** \( t \in \text{transitions from initial state } F.s_0 \text{ with guard } \text{true} \) do

\( n_0 \leftarrow \text{entry}(\text{ICFCs}, t.i) \);

add \((n_0, x_0, t.s_2, \text{null}, t.i)\) to Worklist; \( H[n_0, x_0] \leftarrow x_0 \)

end

**While** Worklist \( \neq \emptyset \) do

remove \((n, x, s, c, i)\) from Worklist; \( y \leftarrow H[n, x] \);

if \( n \) is an API call node then

\( C[n] \leftarrow C[n] \cup \{(c, x)\} \);

**Foreach** \( t \in \text{transitions from } s \text{ with guard } cs = n \text{ or msg } = n \) do

\( n_c \leftarrow \text{entry}(\text{ICFCs}, t.i) \);

\( y_c \leftarrow f_c(y, n, t.i.o) \);

propagate\((n_c, y, y_c, t.s_2, n, t.i)\) ;

end

end

else if \( n \) is an exit of callback then

\( y_r \leftarrow f_{rc}(y, x, c, i.o) \);

\( S_n \leftarrow \text{successors of } s \text{ with transition labeled with } i_{\text{exit}} \);

**Foreach** \( t \in \text{transitions from states of } S_n \text{ with guard } \text{true} \) do

if \( t.i \) is \( \epsilon \) then

**Foreach** \((c_p, x_p) \in C[n] \text{ and } s \in \text{succ}(c) \) do

propagate\((s, x_p, y_r, t.s_2, c_p, t.i)\) ;

end

end

else

\( n_c \leftarrow \text{entry}(\text{ICFCs}, t.i) \);

\( y_c \leftarrow f_c(y_r, c, t.i.o) \);

propagate\((n_c, y_r, y_c, t.s_2, c, t.i)\) ;

end

end

else ... // stays the same as the algorithm in [53]

end
The main data structures of the algorithm include $H$ and Worklist. $H[n, x]$ reports the current solution at node $n \in N$ under context $x \in \mathcal{L}$. The worklist contains 5-tuples $(n, x, s, c, i)$, where $n \in N$ is a node in the ICFGs, $x \in \mathcal{L}$ is the current context for the node, $s \in F.S$ is a state in the CCFA, $c \in N$ is the last API call site we propagated to ($c$ is set to null at the entry node), and $i \in F.X$ is the callback from the CCFA under processing. The data structures Worklist and $H$ are initialized at lines 1-4 for all the callbacks labeled in the transitions from the initial state $F.s_0$. At lines 6–19, we extend the worklist algorithm from [53] to handle two cases of inter-callback propagation: 1) when the current node $n$ is an API call (see lines 7–13), and 2) when the current node $n$ is the exit node of the callback (see lines 14–29). At line 30, the algorithm handles the rest of the cases using the original algorithm [53].

When encountering an API call in the ICFGs, we first identify all the transitions from the current state $s$ that have the guard $cs = n$ or $msg = n$ (at lines 7–9). To preserve the context-sensitivity, at line 8, we save the context and previous call site so that the algorithm can return to the previous call site when exiting from the API call (see line 19). Then, at line 10, using the input of the transition $(t.i)$, which should be the entry point of a callback, we get $n_c$, the entry node in the ICFG of the callback. Our next step is to compute the dataflow facts at the entry node of the callback using transfer function $f_c$. We use a similar approach taken in the inter-procedural analysis. For example, we resolve this references in the callback based on the object receiver of the callback. At line 11, the propagate function adds $(n_c, y, t.s_2, n, t.i)$ to the worklist if the dataflow facts in $y_c$ change the current solution at node $n_c$. The entry node $n_c$ will then be processed when the tuple is retrieved from the worklist.

In the second case, $n$ is the exit node of the callback. At line 15, the transfer function $f_{rc}$ updates the dataflow facts at the return of the callback. At lines 16 and 17, we find all the transitions from the exit point of the current callback. If the input of the transition is epsilon (line 18), we reach the end of an API call. At lines 19–20, we propagate the dataflow $y_r$ to the successors of the call site $c$ based on the context. In another case, at line 24, the transition indicates the start of the
next callback. We first get $n_c$, the entry node of the callback, and then, we compute the updated dataflow facts $y_c$ for the entry node using $f_c$.

We instantiated the dataflow analysis to compute the source and sink pairs of a program, where the source is the definition of a new value and sink is where the value is used. To compute sources and sinks of the program, we instantiated Algorithm 7 for a value flow analysis using a similar approach specified in [24]. Our analysis runs two passes in which the first pass propagates the definition of values through the program (using the modification of the algorithm described in [24]), and the second pass queries the dataflow solution at the node to identify the source statements of each used value.

### 3.4.2 Second Approach

Here, we create a unique main function that simulates all the callback paths found in the CCFA. The analysis works by traversing the CCFA and using branch conditions when there is more than one path from a CCFA state. When a transition in the CCFA is guarded with an API call, we generate a stub method that includes calls to all the callbacks executed inside the API method. Then, we add an edge to the call graph from the API call to the stub method created.

This is a similar approach to the harness methods used in tools such as FLOWDROID. The main difference is that these tools use the lifecycle of components as models to identify the order of callbacks, whereas we generate the harness methods from the CCFAs. The lifecycle of components model provides ordering for callbacks belonging to a component class. For example, they provide the constraint that for an Activity, the callback `onCreate` is called before `onStart`. However, this model does not include the callbacks invoked by external events or in API calls made in the app. For instance, the model used for GUI events in FLOWDROID is not as accurate as the GUI model used in WTGs [63]. On the contrary, CCFA integrates callbacks from external events or API calls. Specifically, we obtain a precise ordering of callbacks regarding Windows and GUI events from the WTG and callbacks invoked in API methods from PCSs. It is worth to mention that API methods can invoke lifecycle callbacks. For example, the API method `startService` invokes
lifecycle callbacks `onCreate` and `onStartCommand` of Services. All these callbacks are summarized in PCSs and included in the CCFA.

The second difference is that for non-lifecycle callbacks, the current implementation of FLOWDROID creates fresh objects instead of reusing the instances defined in the app. This can create unsoundness issues as pointed out by Blackshear et al. [13]. In our approach, for the most cases, the stub methods created from the CCFA use the actual instances created in the apps as the object receiver for each callback. For example, for the app shown in Figure 3.1, our technique creates a stub method for the API method `LoaderManager.initLaoder` (call at line 12) which uses the third parameter as the object receiver for creating a call to the callback `onCreateLoader`. Using this approach, interprocedural analyses, such as the taint analysis defined in FLOWDROID, can analyze each callback with the actual dataflow facts of the callback’s object receiver.

Finally, the harness methods in tools such as FLOWDROID use `null` references for the parameters on the callback calls. In our approach, we use the heuristic of creating unique dummy references for callback calls. This heuristic, while not sound, can help interprocedural analysis to reason about dataflow facts on the parameters of callbacks, which is not allowed on FLOWDROID.

3.5 Evaluation

The goals of our evaluation are to show that 1) the computation of CCFAs is efficient and scalable given a precise call graph, 2) a CCFA can integrate more callbacks in the control flow graphs than the existing solutions, 3) the CCFA is useful for inter-callback analysis of Android apps.

3.5.1 Implementation and Experimental Setup

We implemented our algorithms using Soot [61] and Spark [35] for pointer analysis and call graph construction. To generate the WTG, we extended GATOR [66] and exported WTGs in an XML format which are then loaded and parsed by our tool.
Our study was performed on 135 apps, including 75 real-world apps and 60 benchmarks from DroidBench [60]. The 75 real-world apps include 55 random selected apps from the F-droid repository [39] and 20 apps used in [66], covering 20 out of 30 categories listed on the Google Play Market. When running the 55 F-droid apps, we found that 35 apps ran successfully and 20 apps failed. Specifically, 12 apps crashed when running either Soot, IC3 or GATOR, and 8 apps ran out of time when analyzed with GATOR (we used a timer of 20 minutes). We believe that obfuscation of the APKs is the main factor affecting the reverse engineering process made by tools such as Soot. Additionally, to measure the precision and recall of our inter-callback taint analysis, we selected 60 apps from 4 categories in DroidBench that focus on inter-callback information leaks (the sink and the source are found in different callbacks).

We collected all the API methods called by the apps in our benchmarks. From the list, we removed API methods that likely contain no callbacks until there are 5000 API methods left. We generated PCSs for the 5000 API methods from the Android framework 4.1, among which 133 PCSs contain at least more than one callback.

All of our experiments were run on a virtual machine (VM) with 4 cores and 40GB of memory. We used a 64-bit JVM with a maximum heap size of 20GB. The VM runs on a machine with 16 cores of Quad-Core AMD Operton 6204.

In the following sections, we first report the performance of building CCFAs (Section 3.5.2). We then compare the callback coverage of CCFAs versus WTGs (Section 3.5.3). Next, we demonstrate the improvement of information flow analysis using CCFAs compared to the existing control flow model, the Lifecycle of the components (Section 3.5.4). Finally, we report the results of inter-callback value flow analysis using CCFAs (Section 3.5.5).

### 3.5.2 Performance

In Table 4.3, we report the performance of our tool for computing CCFAs. The first column provides the names of the apps used in our experiments. Under Columns Category, Stmt, CB, WTG, CCFA, we show the category of the app, the number of Jimple statements [61], the total
number of callbacks defined in the app, the number of callbacks covered in the WTG computed for the app, and the number of callbacks covered by the CCFA computed for the app. Column $T$ (s) reports the time (in seconds) used to compute the CCFAs for each app including the time Soot takes to build ICFGs for each callback. As we mentioned in Section 3.3, for each callback, we apply a context-sensitive, flow-sensitive analysis over its ICFG. If the callback was analyzed before under the same context, we copy the previous CCFA returned for the callback and therefore callbacks under the same context are analyzed just once.

Our results show that on average our tool takes 29 seconds to build the CCFA. The performance of the tool depends on the number of API calls made by the app, and since the analysis is flow-sensitive, the performance also depends on the number of different paths with API calls in the app—we refer to the API calls that at least invoke one callback. For apps k9, MyTracks, and XBMC, the analysis ran out of memory because the number of these paths exploded given that we use a context-insensitive pointer analysis to build the call graph of each callback. This can cause an API call to be visited multiple times from different contexts. For these apps, we restricted the number of callees at each call site.

Overall, for most of the real-world apps, our tool builds CCFAs in less than 15 seconds. For bigger apps such as NewsBlur, chann and astrid, our tool finishes in less than 700 seconds. To improve the performance and precision of our tool for big apps, we need more precise pointer analysis and call graph construction algorithms.

### 3.5.3 Callback Coverage

We report the coverage of the callbacks integrated into CCFAs and compared it to WTG—the most advanced callback control flow representation currently available for Android apps. In Table 4.3, Column $CB$ reports the total number of callbacks implemented in the app. Under Columns $WTG$ and $CCFA$, we show the number and the percentage of callbacks integrated into the WTG and CCFA respectively.
### Table 3.1: Performance of Computing CCFAs

<table>
<thead>
<tr>
<th>App</th>
<th>Category</th>
<th>Stat</th>
<th>CB</th>
<th>WTG</th>
<th>CCF</th>
<th>T (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppGen</td>
<td>Navigation</td>
<td>219</td>
<td>9</td>
<td>4</td>
<td>50.00%</td>
<td>8 (100.00%)</td>
</tr>
<tr>
<td>ContactsList</td>
<td>Communication</td>
<td>574</td>
<td>22</td>
<td>10</td>
<td>45.45%</td>
<td>16 (72.73%)</td>
</tr>
<tr>
<td>SDFscanner</td>
<td>Tools</td>
<td>1091</td>
<td>30</td>
<td>5</td>
<td>50.00%</td>
<td>6 (60.00%)</td>
</tr>
<tr>
<td>Obig</td>
<td>Tools</td>
<td>1437</td>
<td>18</td>
<td>5</td>
<td>37.58%</td>
<td>6 (31.58%)</td>
</tr>
<tr>
<td>CamTimer</td>
<td>Photography</td>
<td>2926</td>
<td>50</td>
<td>11</td>
<td>35.00%</td>
<td>11 (65.00%)</td>
</tr>
<tr>
<td>VolDroid</td>
<td>Books</td>
<td>2826</td>
<td>35</td>
<td>11</td>
<td>33.33%</td>
<td>16 (45.71%)</td>
</tr>
<tr>
<td>StandupTimer</td>
<td>Productivity</td>
<td>2835</td>
<td>49</td>
<td>31</td>
<td>63.27%</td>
<td>38 (77.55%)</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>Tools</td>
<td>3114</td>
<td>53</td>
<td>15</td>
<td>28.00%</td>
<td>25 (47.17%)</td>
</tr>
<tr>
<td>TippeyTipsper</td>
<td>Finance</td>
<td>3266</td>
<td>74</td>
<td>48</td>
<td>64.86%</td>
<td>49 (62.22%)</td>
</tr>
<tr>
<td>DrizelleLife</td>
<td>Games</td>
<td>3270</td>
<td>48</td>
<td>28</td>
<td>58.33%</td>
<td>34 (70.83%)</td>
</tr>
<tr>
<td>SuperGenPass</td>
<td>Tools</td>
<td>3177</td>
<td>54</td>
<td>17</td>
<td>34.06%</td>
<td>21 (38.98%)</td>
</tr>
<tr>
<td>BARIA</td>
<td>Tools</td>
<td>3543</td>
<td>60</td>
<td>14</td>
<td>17.50%</td>
<td>19 (23.75%)</td>
</tr>
<tr>
<td>DropboxEditor</td>
<td>Productivity</td>
<td>4269</td>
<td>69</td>
<td>49</td>
<td>71.05%</td>
<td>67 (94.20%)</td>
</tr>
<tr>
<td>HallRail</td>
<td>Navigation</td>
<td>4424</td>
<td>88</td>
<td>14</td>
<td>34.37%</td>
<td>15 (39.47%)</td>
</tr>
<tr>
<td>Reservoir</td>
<td>Productivity</td>
<td>4499</td>
<td>62</td>
<td>37</td>
<td>59.68%</td>
<td>40 (64.52%)</td>
</tr>
<tr>
<td>DroidDroid</td>
<td>Sports</td>
<td>6177</td>
<td>23</td>
<td>11</td>
<td>47.80%</td>
<td>11 (47.80%)</td>
</tr>
<tr>
<td>OpenManager</td>
<td>Tools</td>
<td>4012</td>
<td>49</td>
<td>28</td>
<td>57.14%</td>
<td>34 (69.39%)</td>
</tr>
<tr>
<td>DollDroid</td>
<td>Business</td>
<td>5513</td>
<td>31</td>
<td>30</td>
<td>58.82%</td>
<td>44 (86.27%)</td>
</tr>
<tr>
<td>APV</td>
<td>Productivity</td>
<td>6093</td>
<td>66</td>
<td>37</td>
<td>56.86%</td>
<td>45 (68.18%)</td>
</tr>
<tr>
<td>aXe</td>
<td>Education</td>
<td>6200</td>
<td>51</td>
<td>40</td>
<td>64.76%</td>
<td>40 (67.43%)</td>
</tr>
<tr>
<td>Notepad</td>
<td>Productivity</td>
<td>6411</td>
<td>111</td>
<td>43</td>
<td>66.79%</td>
<td>60 (54.05%)</td>
</tr>
<tr>
<td>JustCalculator</td>
<td>Shopping</td>
<td>6624</td>
<td>85</td>
<td>53</td>
<td>73.83%</td>
<td>41 (91.11%)</td>
</tr>
<tr>
<td>CarCust</td>
<td>Audio</td>
<td>9005</td>
<td>108</td>
<td>63</td>
<td>58.33%</td>
<td>64 (59.26%)</td>
</tr>
<tr>
<td>OpenSteleon</td>
<td>Games</td>
<td>7857</td>
<td>151</td>
<td>43</td>
<td>38.74%</td>
<td>61 (54.95%)</td>
</tr>
<tr>
<td>EPMobile</td>
<td>Medical</td>
<td>12317</td>
<td>161</td>
<td>140</td>
<td>73.70%</td>
<td>141 (79.01%)</td>
</tr>
<tr>
<td>GigaTty</td>
<td>Events</td>
<td>16060</td>
<td>116</td>
<td>36</td>
<td>31.48%</td>
<td>61 (52.59%)</td>
</tr>
<tr>
<td>MFDruid</td>
<td>Video</td>
<td>20454</td>
<td>166</td>
<td>72</td>
<td>43.87%</td>
<td>84 (56.00%)</td>
</tr>
<tr>
<td>L2Up2</td>
<td>Navigation</td>
<td>10172</td>
<td>156</td>
<td>55</td>
<td>35.26%</td>
<td>92 (58.97%)</td>
</tr>
<tr>
<td>SimpleLatex</td>
<td>Audio</td>
<td>12305</td>
<td>98</td>
<td>36</td>
<td>36.75%</td>
<td>42 (42.96%)</td>
</tr>
<tr>
<td>BankerDroid</td>
<td>Tools</td>
<td>15232</td>
<td>35</td>
<td>11</td>
<td>44.00%</td>
<td>14 (56.00%)</td>
</tr>
<tr>
<td>Currency</td>
<td>Finance</td>
<td>14008</td>
<td>84</td>
<td>25</td>
<td>30.96%</td>
<td>34 (41.98%)</td>
</tr>
<tr>
<td>KeyPassDroid</td>
<td>Tools</td>
<td>13396</td>
<td>115</td>
<td>77</td>
<td>66.96%</td>
<td>80 (74.17%)</td>
</tr>
<tr>
<td>MiceDH</td>
<td>Entertainment</td>
<td>16714</td>
<td>156</td>
<td>28</td>
<td>17.93%</td>
<td>54 (33.97%)</td>
</tr>
<tr>
<td>Mileage</td>
<td>Finance</td>
<td>16066</td>
<td>197</td>
<td>78</td>
<td>40.59%</td>
<td>97 (49.24%)</td>
</tr>
<tr>
<td>Cheddar</td>
<td>Games</td>
<td>20177</td>
<td>113</td>
<td>24</td>
<td>21.24%</td>
<td>26 (23.01%)</td>
</tr>
<tr>
<td>NPR</td>
<td>News</td>
<td>24637</td>
<td>130</td>
<td>38</td>
<td>29.23%</td>
<td>76 (53.84%)</td>
</tr>
<tr>
<td>RadioDroid</td>
<td>Audio</td>
<td>25053</td>
<td>175</td>
<td>15</td>
<td>58.33%</td>
<td>36 (20.57%)</td>
</tr>
<tr>
<td>BarcodeScanner</td>
<td>Tools</td>
<td>29087</td>
<td>76</td>
<td>41</td>
<td>53.03%</td>
<td>48 (63.16%)</td>
</tr>
<tr>
<td>Bons</td>
<td>Communication</td>
<td>31253</td>
<td>156</td>
<td>62</td>
<td>32.12%</td>
<td>105 (54.00%)</td>
</tr>
<tr>
<td>SigDroid</td>
<td>Communication</td>
<td>32081</td>
<td>110</td>
<td>40</td>
<td>36.36%</td>
<td>44 (40.00%)</td>
</tr>
<tr>
<td>Axel</td>
<td>Books</td>
<td>32105</td>
<td>90</td>
<td>25</td>
<td>27.78%</td>
<td>30 (44.44%)</td>
</tr>
<tr>
<td>VLC</td>
<td>Audio</td>
<td>34373</td>
<td>570</td>
<td>90</td>
<td>15.79%</td>
<td>107 (18.77%)</td>
</tr>
<tr>
<td>XBMC</td>
<td>Video</td>
<td>34688</td>
<td>436</td>
<td>155</td>
<td>46.13%</td>
<td>167 (49.70%)</td>
</tr>
<tr>
<td>NewsBlur</td>
<td>News</td>
<td>34781</td>
<td>314</td>
<td>70</td>
<td>22.29%</td>
<td>70 (25.16%)</td>
</tr>
<tr>
<td>Connecticut</td>
<td>Communication</td>
<td>36053</td>
<td>255</td>
<td>83</td>
<td>32.33%</td>
<td>140 (43.14%)</td>
</tr>
<tr>
<td>CallRecorder</td>
<td>Tools</td>
<td>43646</td>
<td>116</td>
<td>21</td>
<td>10.11%</td>
<td>31 (15.82%)</td>
</tr>
<tr>
<td>Fbm</td>
<td>News</td>
<td>49162</td>
<td>212</td>
<td>25</td>
<td>11.79%</td>
<td>41 (19.34%)</td>
</tr>
<tr>
<td>Elar</td>
<td>Productivity</td>
<td>63443</td>
<td>502</td>
<td>49</td>
<td>8.26%</td>
<td>67 (11.32%)</td>
</tr>
<tr>
<td>Munch</td>
<td>News</td>
<td>65594</td>
<td>270</td>
<td>24</td>
<td>8.89%</td>
<td>24 (8.89%)</td>
</tr>
<tr>
<td>Domodroid</td>
<td>Tools</td>
<td>68243</td>
<td>266</td>
<td>83</td>
<td>31.20%</td>
<td>110 (40.32%)</td>
</tr>
<tr>
<td>Calculator</td>
<td>Tools</td>
<td>68486</td>
<td>448</td>
<td>27</td>
<td>6.83%</td>
<td>40 (8.93%)</td>
</tr>
<tr>
<td>MyTracks</td>
<td>Navigation</td>
<td>84566</td>
<td>500</td>
<td>99</td>
<td>19.45%</td>
<td>177 (34.71%)</td>
</tr>
<tr>
<td>rhms</td>
<td>Social</td>
<td>100039</td>
<td>535</td>
<td>94</td>
<td>17.57%</td>
<td>117 (21.87%)</td>
</tr>
<tr>
<td>k9</td>
<td>Communication</td>
<td>115219</td>
<td>738</td>
<td>114</td>
<td>15.45%</td>
<td>155 (21.80%)</td>
</tr>
<tr>
<td>nurv</td>
<td>Productivity</td>
<td>143043</td>
<td>1245</td>
<td>199</td>
<td>15.84%</td>
<td>237 (19.04%)</td>
</tr>
</tbody>
</table>

Summary (Avg.) | 25163 | 180 | 38.64% | 48.76% | 29      |
Our results show that on average, WTG integrates 38.04% of the callbacks for an app; the best case coverage is 78.43% (arXiv) and the worst case is 6.03% (Calculator). CCFA integrates 48.76% of the callbacks on average, an increase of 10.72% over WTG due to the consideration of API calls and integration of PCSs and some external events beyond GUI. The best case is 100% (heregps) which include GUI and GPS external events and the worst case is 8.89% (Munch) is mostly composed by fragments. We leave for future work include these callbacks. We found that apps such as astrid and chanu had 149 and 55 different API calls respectively with at least one callback invoked. On the other hand, in small apps such as arXiv we found 0 API calls that invoked at least one callback. These apps contain mostly GUI callbacks (covered by WTG) and callbacks related to external events that CCFAs do not yet include.

3.5.4 Information Flow Analysis

To determine if the CCFAs are correctly built and can be useful, we applied CCFAs for information flow analysis and value flow analysis. In this section, we show that CCFAs can be used to identify inter-callback information leaks in Android apps. Our approach is to integrate CCFAs with the taint analysis offered by FLOWDROID using the second approach described in Section 3.4. We compare the results of FLOWDROID using CCFAs as the inter-callback control flow model against the results of FLOWDROID using lifecycle of components (default implementation). To support inter-component leaks using lifecycle of components in FLOWDROID, we use inter-component models generated by IC3 [47] to enable IccTA [36].

Compared with FLOWDROID on real-world apps. In Table 3.2, we compared the original FLOWDROID with the lifecycle model (see Column Lifecycle) against FLOWDROID integrated with our CCFAs (see Column CCFA). For the apps that are not on the table, both tools reported 0 leaks or ran out of memory. We manually verified each leak reported by both tools and categorized them as true positive (TP) or false positive (FP). For 49 out of 55 apps, both tools generated the same results. FLOWDROID integrated with CCFAs was able to report 33 more leaks than FLOWDROID. We report 2 more true positives for NotePad, 30 more true positives for NPR, and
1 more for Reservator as a result of the CCFAs. The leaks found in these apps were located in callbacks invoked by API calls. The original FLOWDROID creates their own instances to invoke the callbacks in the API methods, which can be unsound [13]. On the other hand, we used the object receivers obtained from the PCSs, which is safer. We were not able to find the leaks found by FLOWDROID in apps Connectbot and Flym because the source or sink was found in callbacks that are not modeled in CCFAs.

Table 3.2: FLOWDROID on lifecycle model and on CCFA

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lifecycle</th>
<th>CCFA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TP</td>
<td>FP</td>
</tr>
<tr>
<td>a2dp.Vol</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>APV</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>arXiv</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>CarCast</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Connectbot</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>DrupalEditor</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Etar</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>Flym</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>Giggity</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>KeePassDroid</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>NewsBlur</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Notepad</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>NPR</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>osmdroid</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>SDScanner</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>SimpleLastfmScrobbler</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>StandupTimer</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>SuperGenPass</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Reservator</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VLC</td>
<td>376</td>
<td>0</td>
</tr>
<tr>
<td>XBMC</td>
<td>276</td>
<td>0</td>
</tr>
</tbody>
</table>

Compared with ground truth given by DroidBench. In the second experiment, we selected 60 apps from DroidBench [60]. DroidBench provides a ground truth on where the security leaks are located, based on which we selected the Lifecycle, InterComponentCommunication, Callbacks and AndroidSpecific four categories that contain inter-callback leaks. The experiment aims to further
measure the true positives and false positives of our inter-callback analysis, and importantly, also to evaluate the false negative rate.

Our results show that our tool is able to detect inter-callback leaks which go across the lifecycle of different components and callbacks related to different classes offered in the Android APIs. The precision and recall of FLOWDROID with CCFAs on these 4 categories were 96.00% and 84.21% respectively, whereas the precision and recall for FLOWDROID with the lifecycle of components were 100% and 80.70% respectively. In Table 3.3, we reported the detailed results of FLOWDROID with the CCFA and FLOWDROID with the lifecycle of components. Under No. of Apps, we show the number of benchmarks in the category we have experimented with. Under TP, FP and FN, we present the number of true positives, false positives and false negatives respectively.

For benchmarks related to the lifecycle of components, including the categories Lifecycle and Callbacks, our tool reported a total of 31 true positives, 1 false positive and 2 false negatives. We reported 1 false positive because our tool does not handle correctly the creation of new objects when the activity is destroyed (e.g. when the phone is rotated). The false negatives were found in benchmarks that have Android constructs (e.g. it Fragments) that we do not yet handle. FLOWDROID is able to handle these apps because it adds calls to non-component classes which enables the taint analysis to find the leaks even though the sequence of callbacks might not be correct. FLOWDROID had 3 false negatives in these two categories due to the use of null references for parameters of callback calls. This is the case in which a parameter of a callback is tainted and then it is used to leak information.

For the InterCompCommunication category, we generated 6 false negatives because our tool, which uses IC3 to resolve inter-component calls, was not able to resolve these particular inter-component calls. We found that even state-of-the-art tools cannot resolve the complex string operations used to define the component. We also generated 1 false negative related to unregistered listeners which neither GATOR nor our tool handles. FLOWDROID with lifecycle of components had the same false negatives regarding unsolved intents because both tools use IC3 for intent resolving, and 1 false negative because of the use of null references in the callback parameters.
Regarding the AndroidSpecific category, we report 1 false positive, which is related to disabled activities that GATOR does not yet model. Both our tool and FLOWDROID reported a false negative in which an external file is tainted with sensitive information and later is read to leak the information.

Table 3.3: Information Flow Analysis on DroidBench

<table>
<thead>
<tr>
<th>Category</th>
<th>Apps</th>
<th>CCFA</th>
<th>FLOWDROID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TP FP FN</td>
<td>TP FP FN</td>
</tr>
<tr>
<td>Lifecycle</td>
<td>17</td>
<td>15 0 2</td>
<td>15 0 2</td>
</tr>
<tr>
<td>Callbacks</td>
<td>15</td>
<td>16 1 0</td>
<td>15 0 1</td>
</tr>
<tr>
<td>InterCommunication</td>
<td>16</td>
<td>8 0 6</td>
<td>7 0 7</td>
</tr>
<tr>
<td>AndroidSpecific</td>
<td>12</td>
<td>9 1 1</td>
<td>9 0 1</td>
</tr>
</tbody>
</table>

3.5.5 Value-Flow Analysis for Source-Sink Pairs

Here, we report our results of value-flow analysis described in Section 3.4. Our goal is to demonstrate that the global, i.e., inter-callback and inter-component value flow do exist in the Android apps, and the definitions (source) of the values are sometimes used (sink) in different callbacks and also different components.

In Table 3.4, under Total, we report the total pairs of sources and sinks found for the app, and under Inter-callback and Inter-comp, we report the pairs where the source and sink are located in different callbacks and different components respectively. We observed that on average, 22.76% of the source-sink pairs were across different callbacks, and the maximum is 62.11% reported by JustCraigslist. We report that 31 out of 55 apps contain inter-component source-sink pairs. We found that in apps such as ContactsList, close to 59% of the inter-callback pairs contain a callback invoked by the API call. This type of global value flows would be missed by tools such as GATOR [66] and also sometimes by FLOWDROID.

For each app, we manually inspected inter-callback source-sink pairs reported (if there are more than 100 pairs, we randomly select 100 pairs). Based on the inspection, we report a false positive rate of 31% on average across all the apps. We found that our tool did not report any false
positives for 16 apps. We also found that there are false negatives because we do not yet model all the libraries where the definitions and uses of the app variables can occur. For example, values can flow from different components by storing data in the objects of classes provided by the Android API (e.g. Intent). Current tools used manual summaries (taint wrappers in [8]) or computed stubs [7] to summarize such dataflow facts.

3.6 Discussions

The CCFA we generated is neither complete nor precise in that the callback invocation paths we obtain from the CCFA only intersect with the actual callback invocation paths. First, the sources we used to generate CCFAs, including WTGs and PCSs together with 9 external events, are not complete. We need to integrate models for other external events and framework constructs such as Fragments. To integrate external events, we need to understand what callbacks are invoked when the events happen, and where the callbacks of external events are registered: whether they are registered through API calls or in the AndroidManifest file loaded when the app starts. Fragments are a part of GUI systems in the Android apps; to integrate Fragments, we need to model how fragments interact with the activity when they are added or removed. In addition, we need to find all the GUI events that are attached to fragments. We plan to continue investigating how to automatically create a model for fragments in our future work.

Second, when constructing CCFAs, Algorithm 5 traverses the loop once, and therefore, we may miss callback invocation paths if there are multiple paths that contain an API call in the loop. Third, the CCFA is a finite state machine, and our current context guards are not yet able to record the state of the window stack. Compared to WTG, we may add infeasible paths that traverse infeasible context of window transitions. To support stacks, a similar approach like k-CFA call graph construction may be applicable, which we will leave to future work.

There are also limitations for using CCFAs for inter-callback analysis. In Algorithm 7, transfer functions $f_c$ and $f_{rc}$ update the dataflow facts when the analysis propagates from callers to callees and from callees to callers respectively. These functions only handle the object receivers but not
Table 3.4: Global Source Sink Pairs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>Inter-callback</th>
<th>Inter-comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>heregps</td>
<td>163</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>ContactsList</td>
<td>99</td>
<td>51</td>
<td>0</td>
</tr>
<tr>
<td>SDScanner</td>
<td>640</td>
<td>224</td>
<td>0</td>
</tr>
<tr>
<td>Obsqr</td>
<td>262</td>
<td>109</td>
<td>0</td>
</tr>
<tr>
<td>CamTimer</td>
<td>1290</td>
<td>353</td>
<td>0</td>
</tr>
<tr>
<td>VxDroid</td>
<td>89</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>StandupTimer</td>
<td>1047</td>
<td>402</td>
<td>0</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>481</td>
<td>47</td>
<td>0</td>
</tr>
<tr>
<td>TappyTipper</td>
<td>537</td>
<td>116</td>
<td>0</td>
</tr>
<tr>
<td>DroidLife</td>
<td>1453</td>
<td>322</td>
<td>3</td>
</tr>
<tr>
<td>SuperGenPass</td>
<td>357</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>BARIA</td>
<td>923</td>
<td>378</td>
<td>43</td>
</tr>
<tr>
<td>DrupalEditor</td>
<td>409</td>
<td>78</td>
<td>5</td>
</tr>
<tr>
<td>HaRail</td>
<td>269</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Reservoir</td>
<td>2641</td>
<td>1076</td>
<td>21</td>
</tr>
<tr>
<td>DiveDroid</td>
<td>1670</td>
<td>158</td>
<td>33</td>
</tr>
<tr>
<td>OpenManager</td>
<td>2969</td>
<td>863</td>
<td>0</td>
</tr>
<tr>
<td>DoliDroid</td>
<td>2487</td>
<td>498</td>
<td>1</td>
</tr>
<tr>
<td>APV</td>
<td>3756</td>
<td>699</td>
<td>2</td>
</tr>
<tr>
<td>arXiv</td>
<td>1249</td>
<td>51</td>
<td>0</td>
</tr>
<tr>
<td>Notepad</td>
<td>1784</td>
<td>500</td>
<td>7</td>
</tr>
<tr>
<td>JustCraigslist</td>
<td>6699</td>
<td>4161</td>
<td>8</td>
</tr>
<tr>
<td>CarCast</td>
<td>3123</td>
<td>541</td>
<td>56</td>
</tr>
<tr>
<td>OpenSudoku</td>
<td>233</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>EPMobile</td>
<td>5037</td>
<td>1808</td>
<td>388</td>
</tr>
<tr>
<td>Giggity</td>
<td>4006</td>
<td>679</td>
<td>2</td>
</tr>
<tr>
<td>MPDroid</td>
<td>896</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>a2dp.Vol</td>
<td>2544</td>
<td>981</td>
<td>75</td>
</tr>
<tr>
<td>SimpleLastfmScrobbler</td>
<td>2971</td>
<td>588</td>
<td>107</td>
</tr>
<tr>
<td>BasketBuildDownloader</td>
<td>322</td>
<td>85</td>
<td>0</td>
</tr>
<tr>
<td>Currency</td>
<td>508</td>
<td>148</td>
<td>0</td>
</tr>
<tr>
<td>KeePassDroid</td>
<td>1018</td>
<td>183</td>
<td>122</td>
</tr>
<tr>
<td>MovieDb</td>
<td>4113</td>
<td>638</td>
<td>262</td>
</tr>
<tr>
<td>Mileage</td>
<td>7174</td>
<td>1765</td>
<td>6</td>
</tr>
<tr>
<td>droidar</td>
<td>4610</td>
<td>305</td>
<td>80</td>
</tr>
<tr>
<td>NPR</td>
<td>2422</td>
<td>517</td>
<td>62</td>
</tr>
<tr>
<td>RadioDroid</td>
<td>626</td>
<td>114</td>
<td>0</td>
</tr>
<tr>
<td>BarcodeScanner</td>
<td>9871</td>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td>Beerin</td>
<td>2235</td>
<td>340</td>
<td>0</td>
</tr>
<tr>
<td>SipDroid</td>
<td>16073</td>
<td>4295</td>
<td>3941</td>
</tr>
<tr>
<td>Aard</td>
<td>613</td>
<td>173</td>
<td>0</td>
</tr>
<tr>
<td>VLC</td>
<td>1466</td>
<td>221</td>
<td>0</td>
</tr>
<tr>
<td>XBMC</td>
<td>5691</td>
<td>1080</td>
<td>12</td>
</tr>
<tr>
<td>NewsBlur</td>
<td>13876</td>
<td>4517</td>
<td>3494</td>
</tr>
<tr>
<td>Connectbot</td>
<td>2373</td>
<td>573</td>
<td>58</td>
</tr>
<tr>
<td>CallRecorder</td>
<td>2480</td>
<td>494</td>
<td>21</td>
</tr>
<tr>
<td>Flym</td>
<td>1619</td>
<td>408</td>
<td>15</td>
</tr>
<tr>
<td>Etar</td>
<td>17544</td>
<td>2803</td>
<td>14</td>
</tr>
<tr>
<td>Munch</td>
<td>2177</td>
<td>315</td>
<td>13</td>
</tr>
<tr>
<td>Domodroid</td>
<td>12391</td>
<td>434</td>
<td>0</td>
</tr>
<tr>
<td>Calculator</td>
<td>5782</td>
<td>1155</td>
<td>0</td>
</tr>
<tr>
<td>MyTracks</td>
<td>5737</td>
<td>1138</td>
<td>421</td>
</tr>
<tr>
<td>chamu</td>
<td>13147</td>
<td>1207</td>
<td>266</td>
</tr>
<tr>
<td>k9</td>
<td>12558</td>
<td>2264</td>
<td>1261</td>
</tr>
<tr>
<td>astrid</td>
<td>27329</td>
<td>10959</td>
<td>3045</td>
</tr>
</tbody>
</table>

Summary (Avg.) 4088 930 (22.76%) 251 (6.16%)
the return values and input parameters of the callbacks. Also, during the analysis of API methods, we may encounter an unknown object receiver for the callbacks (inherit from imprecisions in the PCSs). If the callbacks are not invoked in the API methods, we create a global variable as object receivers instead of using the actual instances created in the app, which is neither precise nor safe. Also, as mentioned before, our inter-callback analysis has not yet modeled the effect of the event-queue for asynchronous calls.

In addition, we have not yet used the guards labeled on the transitions of CCFAs in our client analyses. We foresee that the guards can be used for test input generation and dynamic race detection, where the proper events and messages are needed to be setup for executing a path of interest. The guards can also be used for infeasible path detection on CCFA, which can help static analyses like the information flow and value flow analyses presented in this chapter. We plan to implement such improvement in the future work.

3.7 Summary

This chapter presents the definition, construction and applications of a new representation, callback control flow automata (CCFA) for specifying control flow of callbacks in the event-driven, framework-based systems. The representation can express sequential as well as calling relationships between callbacks and can specify both synchronous and asynchronous callback invocations together with the conditions under which these invocations are triggered. We implemented the tool to automatically generate CCFAs from the Android apps’ code, and integrated the CCFAs with existing data-flow analysis to support inter-callback analysis. Our experimental results show that the construction of CCFAs is efficient and scalable, we improved the callback coverage over WTGs, and CCFAs improved results in detecting information leaks. Our results also indicate that inter-callback dataflow commonly exists, and callback control flow modeling is important for analyzing and testing such global behaviors.
CHAPTER 4. TESTING COVERAGE CRITERIA FOR MOBILE APPS
BASED ON CALLBACK SEQUENCES

In recent years, several tools have been developed to detect faults in Android apps using automatic testing [29, 45, 58, 41, 9, 5, 20]. The majority of these tools focus on the analysis and testing of Graphical User Interfaces (GUI) [34]. These approaches work on generating tests that maximized a test criterion based on GUI event sequences and also the improvement of a structural coverage metric such as statement coverage.

We believe that such criteria do not completely address the adequacy of test suites for mobile apps. Mobile apps, besides a rich GUI, also use other constructs to accomplish tasks. For example, background tasks executed through Services or AsyncTasks are not included in the GUI models. Moreover, the callbacks of these background tasks can interleave with GUI and other external events callbacks. Callbacks of other external events, such as GPS location updates or sensors, can also interleave with GUI callbacks. A coverage criterion based on GUI event sequences is not able to distinguish all the possible interleavings of callbacks that can be executed at run time. To test such code, we need to consider not just the execution behavior from the user through GUI, but also from the API calls to the framework and external components such as the camera or the GPS on the phone.

In this chapter, we introduce coverage criteria for testing Android apps based on the execution of callbacks. Specifically, we consider coverage criteria based on callback sequences, as they represent the behaviors that occur between different components of the mobile phone. For instance, we can recognize different behaviors by covering callback interleavings between event handlers (including any kind of event such as GUI, location, etc.), asynchronous and synchronous callbacks invoked in API methods [49]. Thus, a coverage criterion based on different types of callbacks can cover
behaviors such as the execution of background tasks using Services, an advantage over black-box coverage criteria.

Before developing coverage criteria based on callbacks, we need to address some challenges. The number of possible sequences of callbacks can be intractable to compute. We evaluate what types of callbacks are important and restrict the length of sequences to compute. We perform an empirical study to understand how the data flows between callbacks. We use this insight to define what types of callbacks more commonly share data and design coverage criteria to test the interactions of these callbacks. Additionally, we develop a bug study to find what types of callbacks may introduce buggy behavior. Based on these two factors, we designed 3 callback coverage criteria for Android apps. To compute the different callbacks sequences, we use a static representation called Callback Control Flow Automata (CCFA) [50]. This model allows computing precise callbacks sequences considering callbacks invoked asynchronously and synchronously from external events and in API methods.

We design 3 coverage criteria based on callback sequences, named C1, C2, and C3. C1 is designed to cover callback interactions related to event callbacks. That include GUI event handlers and other external event handlers such as callbacks that react to GPS and sensors events. Coverage criterion C2 is designed to cover sequences of callbacks that involve callbacks invoked in API methods synchronously. Lastly, we design coverage criterion C3 to detect different concurrent behaviors in the app including callbacks that are invoked asynchronously in API methods.

We develop algorithms to generate callback sequences from the CCFA for C1, C2, and C3 and instrument apps to generate callback traces (it logs the execution of entry and exit points of callbacks). To measure coverage of each criterion, we compare whether the callback traces generated from the tests contain each of the callback sequences. These results in a total number of sequences each test cover. The coverage can be computed by dividing the number of callbacks sequences covered over the total number of sequences generated for each criterion.

This chapter makes the following contributions:

- An empirical study to understand the data flow behavior between callbacks
- The design of 3 coverage criteria based on callbacks sequences
• Algorithms to compute the coverage criteria from a static model, the CCFA
• Empirical evidence that shows the importance of our coverage criteria based on real faults found in apps

4.1 Motivation

In this section, we use the real-world bugs found in the apps *chanu* (issue #140[16]) and *FileDownloader* (issue #610[2]) to show the importance of callback-sequences based testing criteria.

The first example presents an issue found in the app *chanu* which involves a GUI event and the camera [16]. Figure 4.1a shows the three callbacks in an activity that takes a picture in *chanu*, including `onPictureTaken()`, `onResume()` and `onClick()`. When the user clicks the GUI button, the system executes the callback `onClick()` which then calls the API method `takePicture()` (at line 18). This method triggers an asynchronous task to capture an image and execute the callback `onPictureTaken()`. Figure 4.1b shows the control flow graph that models the possible sequences of these callbacks. Along path `onClick() → onClick()`, the user double-clicks a button, while path `onClick() → onPictureTaken() → onClick()` models the scenario where the user clicks the button, waits until the picture is taken, and then tabs the button again.

The bug is found along path `onClick() → onClick()`. When the framework executes the callback `onClick()` for the second time before `onPictureTaken()` is invoked, the API method `takePicture()` crashes with a runtime exception. The root cause is that the camera is busy responding to the first `onClick()`, and there is a race condition on a global flag in `takePicture()`. Whereas, along path `onClick() → onPictureTaken() → onClick()`, `takePicture()` has finished `onPictureTaken()` corresponding to the first click and can proceed with the second click. To fix this bug, the developer added a flag `mTaken` to avoid calling `takePicture()` while the camera is busy (see the fixes at lines 5, 17, 19 and 20 in Figure 4.1a). If the testing only aims to cover all statements or all callbacks, we can achieve 100% coverage, stop testing after covering the path `onResume() → onClick() → onPictureTaken()`, and miss the bug. Event-based testing models [46] do not identify the paths with the following callbacks `onClick() → onClick()` and `onClick() →
onPictureTaken() \rightarrow onClick(), as the event-based model does not include the callbacks invoked in the API. Thus, it may also miss the bug.

In the second case, we show a bug (the issue #610 in the app FileDownloader [2]) that only can be triggered when we consider different callback sequences in the API method. In Figure 4.1c, we show a simplified fragment of a Service that uses the SQLite APIs provided by Android framework to access a database. When the API method `getWritebleDatabase()` is called at line 5, it prepares the database $db$ and then returns $db$. There are a set of paths implemented in this API method, shown in Figure 4.1d. The path `FDService.onCreate() \rightarrow DBHelper.onCreate()` represents the case where the app is being freshly installed. The path `FDService.onCreate() \rightarrow DBHelper.onUpgrade()` represents the case where the app is updating an old version. The bug is located on the second path. In `FDService.onCreate()` at line 4, the constructor `DBHelper()` (see its implementation at line 9) is invoked. The function turns on "logging" functionality to maintain a copy of cache for the database. The bug is found along the path of callbacks `FDService.onCreate() \rightarrow DBHelper.onUpgrade()` which updates the database schema in `DBHelper.onUpgrade()` but is not visible immediately because the use of `setWriteAheadLoggingEnabled()` made at line 10.

In this case, there are two different behaviors that happen after calling `getWritebleDatabase()` depending on the environment in which the app is run. If the app has a fresh install, then the test would cover just one callback sequence. To detect the bug, we need to configure the environment where the app is installed and updates a previous version already installed in the system. Event-based criteria does not cover such behavior.

To find the bugs in the above scenario, we need test criteria that can consider different sequences of callbacks such as callback sequences between GUI and API method, and the callback sequences invoked along different paths of API methods.

### 4.2 Designing Test Criteria Based on Callback Interactions

In the previous section, we have shown that it is important to consider callback sequences along different paths during testing. In this section, we perform 2 studies to understand how important
class CamAct extends Activity {
    PictureCallback mP = new PictureCallback() {
        void onPictureTaken(byte[] d, Camera camera) {
            ... + mTaken = true;
        }
    };
    void onResume() {
        cam = getCameraInstance();
        ... + CaptureList l = new CaptureList();
        captureButton.setOnClickListener(l);
    }
}

class CaptureList extends OnClickListener {
    void onClick(View v) {
        if (mTaken) {
            cam.takePicture(null, null, mP);
            mTaken = false;
        }
    }
}

(a) Issue #140 reported in chanu \[16\]

class FDService extends Service {
    void onCreate() {
        ... DBHelper dbh = new DBHelper(this);
        db = dbh.getWritableDatabase();
    }
}

class DBHelper extends SQLiteOpenHelper {
    DBHelper(Context context) {
        setWriteAheadLoggingEnabled(true);
    } void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE ...");
    } void onUpgrade(SQLiteDatabase db, ...) {
        if (oldVersion < 2) {
            String addColumn = "ALTER TABLE " ...
            db.execSQL(addColumn);
        }
    }
}

(b) Sequences of callbacks executed in chanu

class DBHelper extends SQLiteOpenHelper {
    DBHelper(Context context) {
        setWriteAheadLoggingEnabled(true);
    } void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE ...");
    } void onUpgrade(SQLiteDatabase db, ...) {
        if (oldVersion < 2) {
            String addColumn = "ALTER TABLE " ...
            db.execSQL(addColumn);
        }
    }
}

(c) Issue #610 reported in FileDownloader \[2\]

d) Sequences of callbacks executed in FileDownloader

Figure 4.1: Motivating examples using real world apps chanu and FileDownloader. We present a
fragment of the code that is significant to the bug and the possible sequences of callbacks in the
fragment that can be executed at runtime
to cover these interactions during testing. Then, we designed a set of test criteria based on our findings.

4.2.1 Specifying Callback Sequences Using CCFA

As we mentioned in Chapter 3, event-driven, framework-based mobile apps base their implementation on callbacks provided by the Android framework's API methods. When an app is running, the framework executes the callbacks depending on the external events and the API calls made by the app. The coverage criteria we define is based on callback sequences. To generate the possible callback sequences for each coverage criterion we define, we implement different algorithms on top of the CCFA. We describe these algorithms in Section 4.3.2.

4.2.2 What important callback sequences we should test?

To answer this question, we designed two studies. In the first study, we investigate whether there exists dataflow between callbacks and what types of callbacks share dataflow. We believe that if there commonly exists dataflow between two callbacks, we need to test the inter-callback paths, as any incorrect definition or use of variables may only be exposed through exercising the inter-callback paths. In the second study, we explore whether there are bugs whose root causes are related to callback interactions and what types of callback interactions are likely buggy. We need to especially test such interactions of callbacks to avoid similar bugs in different apps.

4.2.2.1 Study 1: Does there exist dataflow between callbacks? What types of callbacks share dataflow?

We performed a def-use analysis for all the variables found on the CCFA on 55 apps randomly selected from the Google Play Market\(^1\) (same apps from Chapter 3). We report that an *inter-callback dataflow* is found when there is a definition whose use is in a different callback. Our results show that all 55 apps contain inter-callback dataflow. Among all the def-use pairs we computed, 22.76\% are inter-callback.

\(^1\)https://play.google.com
We performed further analysis to identify the type of callbacks in which def-use pairs are located. We found that 55% of the inter-callback dataflow occurs between the callbacks that respond to external events, namely *event callbacks*. This is because the majority of the apps contain functionalities related to the GUI or the sensor components such as GPS or camera. The second most common (34% of) inter-callback dataflow is located between synchronous callbacks invoked by the API method, namely *API SYNC callbacks*. This is because many API methods contain more than one synchronous callbacks, and it is common to share data in an API method. We also found def-use pairs between event callbacks and callbacks invoked in API methods, specially 7% between event and APSYNC callbacks and 6% between event and APIASYNC callbacks. APIASYNC are asynchronous callbacks invoked in API methods, typically for responding to the *messages* from the API method. This interaction represents the scenario where the event handler invokes an API method and passes the data to the API method for handling the event. See Table 4.1 for the summary of our results.

Table 4.1: Dataflow between callbacks. EV: event callback, AS: API_SYNC callback, AA: API_ASYNC callback

<table>
<thead>
<tr>
<th></th>
<th>EV-EV</th>
<th>AS-AS</th>
<th>EV-AS</th>
<th>EV-AA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>55%</td>
<td>34%</td>
<td>7%</td>
<td>6%</td>
</tr>
</tbody>
</table>

In addition, our study reports that although the global dataflow can propagate through a maximum of 18 callbacks, 38% global dataflow are related to consecutive callbacks sequenced on CCFA.

### 4.2.2.2 Study 2: What types of callbacks are likely buggy?

As a pilot study, we analyzed 1526 bugs from 6 apps. We identified the bugs that lead to a crash in the app and found that 26 of such bugs are related to callback interactions. That is, we need to exercise a sequence of callbacks to trigger the crash. Furthermore, we found that all the 26 bugs are related to 2 callbacks. 85% of the bugs are related to the two consecutive callbacks
specified in the CCFA. This data aligns with our findings from the first study that many of the
time, there is data sharing at the neighbor callbacks in the execution paths.

Our results are presented in Table 4.2. The first column shows the apps we studied. Under
*Bugs*, we show the number of bugs we inspected. Under *Multi-C*, we give the number of bugs whose
root causes are related to multiple callbacks. Under *EV-EV, AS-AS, EV-AS* and *EV-AA*, we show
what types of callback interactions that lead to the bug. Our results show that there are 6 bugs
related to interactions of two event handlers, 11 bugs are related to synchronous calls in the API
method, and 13 bugs are caused by improper interactions between the event handler callbacks and
the asynchronous callbacks in the API method. These bugs are mostly race conditions as a result
of asynchronous invocations.

Table 4.2: Bug study results. EV: event callback, AS: API_SYNC callback, AA: API_ASYNC
callback

<table>
<thead>
<tr>
<th>App</th>
<th>Bugs</th>
<th>Multi-C</th>
<th>EV-EV</th>
<th>AS-AS</th>
<th>EV-AS</th>
<th>EV-AA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectBot</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>FileDownloader</td>
<td>12</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>AntennaPod</td>
<td>133</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>cgeo</td>
<td>273</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Wordpress</td>
<td>315</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Ankidroid</td>
<td>784</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1526</strong></td>
<td><strong>26</strong></td>
<td><strong>6</strong></td>
<td><strong>7</strong></td>
<td><strong>4</strong></td>
<td><strong>13</strong></td>
</tr>
</tbody>
</table>

4.2.2.3 Conclusions of the Studies.

From the study, we learned that although it is beneficial to test long callback sequences, the
priority is to thoroughly cover neighbor callback interactions, starting with a length of two callbacks.

First, we want to test interactions between event callbacks as these callbacks share data and
they occupy the main behavior of apps. Second, synchronous callbacks invoked in the API methods
are typically not targeted by any GUI testing tools; however, there are data sharing and many bugs
related to such callbacks. We want to trigger these callbacks in testing. Finally, the main source
of callback interaction bugs goes to race conditions caused by asynchronous invocations. We need
to well test the interleavings of relevant callbacks. In the following, we show the testing criteria we designed based on these findings.

4.2.3 Testing Criteria Based on Callback Sequences

Here, we present three testing criteria, namely $C_1$, $C_2$ and $C_3$. In the following, we first formally introduce the concept of callback sequence. It is used to define the three criteria.

**Definition 2 (Callback Sequence).** Let $C = \{c_1, ..., c_n\}$ be the callbacks implemented for an app $P$. The input symbols of the CCFA for $P$ are defined by the set $I = \{c_{1\text{entry}}, c_{1\text{exit}}, ..., c_{n\text{entry}}, c_{n\text{exit}}\}$. A callback sequence is $\langle s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_m \rangle$ where $s_i \in I$ for $1 \leq i \leq m$. The length of a callback sequence is the number of callbacks $c$, or in another words, the number of the input symbol $c_{\text{entry}}$ the sequence contains.

4.2.3.1 Coverage Criterion $C_1$

Coverage criterion $C_1$ is designed to cover callback interactions related to event callbacks. The events can be a GUI event or the events from the sensing components.

**Definition 3 ($C_1$).** Let $C = \{c_1, c_2, ..., c_n\}$ the callbacks implemented for an app $P$. Let $E \subseteq C$ be the set of callbacks for handling events in an app $P$. Let $E_I$ be the set of input symbols (the entry and exit points) for any callback $c \in E$. A coverage criterion $C_1$ is satisfied if and only if for all callback sequences of length 2, $\langle s_1 \rightarrow s_2 \rightarrow s_3 \rangle$, where $s_1, s_2, s_3 \in E_I$, generated from the CCFA of $P$, the traces $T$ contain $S$.

The behaviors required to be covered by $C_1$ are close to the behaviors covered by GUI testing tools [5]. The difference is that $C_1$ also requires to exercise the interactions between callbacks invoked in other types of external events besides GUI events. As an example, in Figure 4.2a, the app registers the listener callbacks for two buttons at lines 7 and 8, and a callback to receive GPS location updates at line 9. Figure 4.2b shows the CCFA built for the app. Given $C_1$, we generate the following sequences of callbacks for $A.onCreate()$:
```java
class A extends Activity {
    Button b1;
    Button b2;
    LocList l;

    void onCreate(Bundle b) {
        ..
        b1.setOnClickListener(new Button1());
        b2.setOnClickListener(new Button2());
        lm.requestLocationUpdates(..., new LocList());
    }
}

class LocList implements LocationListener {
    void onLocationUpdate(...) {
        ...
    }
}

class Button1 extends OnClickListener {
    void onClick(View v) { ... }
}

class Button2 extends OnClickListener {
    void onClick(View v) { ... }
}
```

(a) Source code of a simple Android app including the GPS event

(b) A CCFA for the app

Figure 4.2: An app with Location Callbacks

1. A.onCreateentry → A.onCreateexit → Button1.onClickentry
2. A.onCreateentry → A.onCreateexit → Button2.onClickentry
3. A.onCreateentry → A.onCreateexit → LocList.onLocationUpdateentry

Using a similar approach, we can generate the callback sequence centered on Button1.onClick (including 2 sequences), Button2.onClick (2 sequences) and LocList.onLocationUpdate (2 sequences). As a result, we have a total of 9 (3+2+2+2) callback sequences to cover. To satisfy C1 for this app, we need to ensure that testing covers the cases where the location events happen before or after a GUI callback. Whereas, any GUI based coverage criterion just need to cover the combination of the GUI callbacks.
4.2.3.2 Coverage Criterion C2

Coverage criterion C2 focuses to trigger the callback interactions related to synchronous callbacks implemented in the API methods (APLSYNC). Specifically, there are two types of sequences that we aim to test: 1) a synchronous callback in the API method and the caller of the API method which is an event callback, and 2) two synchronous callbacks in the same API method. For example, in Figure 3.1b, we aim to test the sequence $A.onStart_{entry} \rightarrow L.onCreateLoader_{entry}$. In Figure 4.1d, we will test $DBHelper.onCreate_{exit} \rightarrow DBHelper.onOpen_{entry}$ and $DBHelper.onUpgrade_{exit} \rightarrow DBHelper.onOpen_{entry}$. All these callbacks are synchronous callbacks implemented in the API method `dbh.getWritableDatabase()` at line 5 in Figure 4.1c.

**Definition 4 (C2).** Let $Z = \{c_1, c_2, ..., c_n\}$ and $Z \subseteq C$ be the set of callbacks of type APLSYNC in an app $P$, and $Z_I$ be the set of input symbols (the entry and exit points) for any callback $c \in Z$. Let $E \subseteq C$ be the set of callbacks for handling events in an app $P$, and $E_I$ be the set of input symbols (the entry and exit points) for any callback $c \in E$. A coverage criterion C2 is satisfied if and only if for all call sequences of length 2, $(s_1 \rightarrow s_2)$, where $s_1 \in E_I$ or $s_1 \in Z_I$ and $s_2 \in Z_I$, generated from the CCFA of $P$, the traces $T$ contain $S$.

This test criterion requires to trigger the API method as well as the synchronous call(s) in the API method. Sometimes, there exist different callback sequences in the API method dependent on the state of the system when the API call is made. C2 requires to test all possible callback sequences. In Figures 4.1c and 4.1d, we will test all the following callback sequences from `FDService.onCreate()`:

1. `FDService.onCreate_{entry} \rightarrow DBHelper.onCreate_{entry}`

2. `FDService.onCreate_{entry} \rightarrow DBHelper.onUpgrade_{entry}`

3. `FDService.onCreate_{entry} \rightarrow DBHelper.onOpen_{entry}`

The states that can trigger these callback sequences include (1) the app is just installed, (2) the app is just updated from an older version previously installed on the phone, and (3) the app is
running without any installation and upgrading. That is, in this example, we even need to prepare special environments to satisfy C2.

4.2.3.3 Coverage Criterion C3

The API methods can make asynchronous calls such as `Handler.sendMessage` and `Context.startService`. These methods will invoke the asynchronous callbacks (we call them API_ASYNC). The past studies [43, 32, 11] as well as our own bug study presented in Section 4.2.2 all indicate that asynchronous callbacks invoked in the API calls have lead to concurrency bugs. Such bugs typically involve two event calls and one API_ASYNC. For example, in Figures 4.1a and 4.1b, there are two possible interleavings between the callbacks of `CaptureList.onClick()` and `CamAct$3.onPictureTaken`. See the following for the actual execution traces resulted from the two interleavings:

1. `CaptureList.onClick_entry → CaptureList.onClick_exit → CamAct$3.onPictureTaken_entry → CamAct$3.onPictureTaken_exit → CaptureList.onClick_entry`

2. `CaptureList.onClick_entry → CaptureList.onClick_exit → CaptureList.onClick_entry → CaptureList.onClick_exit → CamAct$3.onPictureTaken_entry`

In the first case, there are no other tasks in the event queue when the task of `CamAct$3.onPictureTaken()` was posted by `takePicture()`. As a result, `CamAct$3.onPictureTaken()` is executed immediately after `CaptureList.onClick()`. In the second case, the second `CaptureList.onClick()` follows right after the first `CaptureList.onClick()` before `takePicture()` posts the task of `CamAct$3.onPictureTaken()`. The implementation of the app does not consider such case and will lead to a crash in the API method. We should test such interleavings to help expose the problems.

We developed an algorithm that takes the first API_ASYNC (named c) in an API method as an input and traverses the CCFA to find its caller (named e). We identify any successor of e on CCFA, s, which is also an event callback. s and c potentially run concurrently. The algorithm then
returns all the interleavings regarding e, s and c. We focus on the interleaving between two event callbacks and one API_ASYNC, as most of the bugs we found are caused by such interleavings.

**Definition 5 (C3).** Let $Y = \{c_1, c_2, ..., c_n\}$ and $Y \subseteq C$ be the set of API_ASYNC callbacks. Let $Y_I$ be the set of input symbols (the entry and exit points) for any callback $c \in Y$. A coverage criterion $C3$ is satisfied if and only if for every callback sequence $S$ of length 3 generated from the CCFA for each $c \in Y$, $T$ contains $S$.

It should be noted that C1 also tests sequences involving two event callbacks. However, C3 is stronger to enforce all the interleaving between two event callbacks and a API_ASYNC callback. C1 only requires to cover two events without enforcing any behavior related to API_ASYNC.

### 4.3 Measuring Test Coverage

In this section, we present our methodologies of measuring the test coverage of our coverage criteria C1, C2, and C3.

#### 4.3.1 The Framework for Measuring C1, C2 and C3

As shown in Figure 4.3, given an Android app (in the APK format) and its tests, we first generate the CCFA for the app. Using the CCFA, we apply different traversal algorithms (described later in this section) to generate callback sequences based on the coverage criteria C1, C2, and C3. These are the callback interactions an ideal test set should cover for the app. We also analyze the CCFA.
to identify whether a callback is an event callback, API\_SYNC or API\_ASYNC. On the CCFA, the transitions of an event callback are labeled with its triggering event, the transitions labeled with the message represent the API\_ASYNC (these calls are triggered by the messages in the Android framework), and finally for any callbacks invoked in the API method (labeled with the API call site between the two $\epsilon$ edges on the CCFA), they are the API\_SYNC if not already marked as API\_ASYNC. This mapping between the callbacks and their types will be used for analyzing the trace to determine what callback interactions a test actually covers.

To collect the trace, we develop a tool that automatically instruments all the callbacks in the app. Our tool takes an APK file and first identifies all the callbacks implemented in the app. Then, for each callback, our tool instruments the entry and exit points of the callback. For entry points, our tool finds the first statement in the control flow graph (CFG) of each callback and insert instrumentation before them. For exit points, our tool finds all the return statements in the callback and adds instrumentation before each return statement. Each line in the trace prints a tuple $(t, s, k)$, where $t$ is the time in milliseconds when the instrumented statement was executed, $s$ is the signature of the callback and $k$ is either ENTRY or EXIT to represent the entry and exit points respectively.

In this work, we have used automatically generated test inputs to run the apps. In Android, like in other event-driven systems, most of the automated test input generation tools focus on randomly generating external events [29] or using some black-box models [58, 45, 5]. The approach is to run very long sequences of events in each test and execute a test for a long period of time, so they have a chance to reach the states of the app that are typically hard to reach. In previous studies, the researchers set up tools to run from 1 hour to 3 hours, dependent on the sizes of the apps, for testing, and generated one very long trace [22, 62, 45, 58]. We adopted a similar approach.

After testing with the instrumented app, we collect the trace $T$, consisting of a sequence of entry and exit points of callbacks executed. For example, a test that launches the app and then executes a back button event for the app shown in Figure 3.1a generates the following trace (named $T_{3.1a}$):
In the following, we provide further details regarding how we measure C1, C2 and C3, specifically how we analyze the CCFA to identify what callback sequences the test should cover for a particular app, how we analyze the trace collected during testing to obtain what the test has actually covered, and finally how we calculate the actual coverage based on the two.

### 4.3.2 Measuring Coverage for C1

To compute what we should test for C1, we traverse the CCFA of the app and find all the two consecutive event callbacks (we name this set $S_{c1}$). To do so, we first modify the CCFA and remove all the transitions between the $\epsilon$ transitions. The $\epsilon$ transitions mark the beginning and the end of the API calls. Thus this step removes all the API\_SYNC and API\_ASYNC callbacks in the CCFA, so we can consider only event callbacks which are needed for C1. Using the modified CCFA, we start at every transition that represents the entry point of a callback, and generate sequences of two consecutive callbacks. For loops, we traverse the loop once.

As an example, in Figure 3.1b, we traverse the transitions from $q_1$ to $q_4$ and generate the sequence $A.onCreate_{entry} \rightarrow A.onCreate_{exit} \rightarrow A.onStart_{entry}$. Based on the $\epsilon$ transitions at $q_4$ to $q_6$ and $q_8$ and $q_9$, we remove the callbacks in the API methods, including all the transitions from $q_6$ to $q_9$. As a result, we generate the sequences $A.onStart_{entry} \rightarrow A.onStart_{exit} \rightarrow CList.onClick_{entry}$ and $A.onStart_{entry} \rightarrow A.onStart_{exit} \rightarrow A.onStop_{entry}$ by taking either path $q_3 \rightarrow q_4 \rightarrow q_5 \rightarrow q_{10}$ or $q_3 \rightarrow q_4 \rightarrow q_5 \rightarrow q_{11}$. 
To measure coverage criteria C1, we compare the sequences computed from the CCFA with the traces. We first filter the trace to contain just event callbacks using the mapping of callbacks and their types pre-computed from the CCFA, resulting in the trace $T_E \subseteq T$. We then check whether a sequence $s \in S_{C1}$ is included by $T_E$. For instance, the example trace $T_{3.1a}$ shown above does not contain the sequence $A.onStart_{entry} \rightarrow A.onStart_{exit} \rightarrow A.onStop_{entry}$. When we compute event callback traces for C1, we generate:

1. $(t1, A.onCreate(), ENTRY)$
2. $(t2, A.onCreate(), EXIT)$
3. $(t3, A.onStart(), ENTRY)$
4. $(t6, A.onStart(), EXIT)$
5. $(t7, A.onStop(), ENTRY)$
6. $(t8, A.onStop(), EXIT)$

It indicates that $A.onStart_{entry} \rightarrow A.onStart_{exit} \rightarrow A.onStop_{entry}$ is covered. Let $C_{C1} \subseteq S_{C1}$ be the set of sequences actually covered by $T_E$. The coverage C1 for the test is computed by $|C_{C1}|/|S_{C1}|$. Note that when there are multiple traces generated in testing, $C_{C1}$ will include the sequences covered by all the traces.

### 4.3.3 Measuring Coverage for C2

To compute what we should test for C2, we focus on the callbacks invoked synchronously in the API methods. Thus our first step is to remove all the asynchronous callbacks invoked in the API method (the transitions labeled with the message guard). We then traverse the modified CCFA. When the traversal reaches the transition labeled with the entry point of an API_SYNC, it follows the edges on the CCFA backward to identify the entry points of its predecessor callback or its caller callback. For example, in Figure 3.1b, when we reach the transition from $q_6$ to $q_7$, we perform a backward traversal to identify the entry point of the calling callback $A.onStart_{entry}$. Thus, for the transitions $q_3 \rightarrow q_4 \rightarrow q_6 \rightarrow q_7$, we generate the callback sequence $A.onStart_{entry} \rightarrow L.onCreateLoader_{entry}$. Using a similar way, the traversal visits all the synchronous callbacks in the API methods and gets their predecessors to form the sequences.
To determine what callback sequences required by C2 have been covered by the trace $T$, we check if any of the required sequence computed above $s \in S_{C2}$ is covered by $T$. Note that in the trace, the two callbacks of API_SYNC should always occur consecutively if they are consecutive on the CCFA, so we do not need to filter out any callbacks before the checking.

Let $C_{C2} \subseteq S_{C2}$ be the set of sequences covered by $T$. Then, the coverage $C2$ for the test is $|C_{C2}|/|S_{C2}|$.

### 4.3.4 Measuring Coverage for C3

C3 requires to test different interleavings involving the APIASYNC callbacks. To compute all the required callback sequences $S_{C3}$, we first traverse the CCFA and find the entry points of the APIASYNCs. For example, in Figure 4.4 (see the CCFA for the code shown in Figure 4.1a in Section 4.1), we first identify the callback, onPictureTaken() indicated by the transition $q_5 \rightarrow q_6$ (we called this transition $t$). Our next step is to find the caller of such callback. This goal is
achieved by traversing the CCFA backwards until we find an asynchronous transition. For the
transition \( t = q_5 \rightarrow q_6 \), we identify \( \text{CaptureList.onClick} \) from the transition \( q_3 \rightarrow q_4 \) (we call
this transition \( t_{\text{caller}} \)).

To find the asynchronous callbacks that potentially have a race condition with \( t \), we start
traversing the CCFA at \( t_{\text{caller}} \), and find all of its possible asynchronous callback successors. These
are callbacks that can follow \( t_{\text{caller}} \), and dependent on the timing, they can be executed before \( t \) or
after \( t \), creating different interleavings among these callbacks. For example, given \( t_{\text{caller}} \) found at
\( q_5 \rightarrow q_6 \) in Figure 4.4, we find the next reachable asynchronous callbacks \( \text{CaptureList.onClick} \) and
\( \text{CaptureList.onTab} \) from the transitions \( q_3 \rightarrow q_4 \) and \( q_3 \rightarrow q_9 \) respectively. Based on which we can
generate the following four callback sequences:

1. \( \text{CaptureList.onClick.entry} \rightarrow \text{CaptureList.onClick.exit} \rightarrow \text{CamAct$3.onPictureTaken.entry} \rightarrow \text{CamAct$3.onPictureTaken.exit} \rightarrow \text{CaptureList.onClick.entry} \)

2. \( \text{CaptureList.onClick.entry} \rightarrow \text{CaptureList.onClick.exit} \rightarrow \text{CaptureList.onClick.entry} \rightarrow \text{CaptureList.onClick.exit} \rightarrow \text{CamAct$3.onPictureTaken.entry} \)

3. \( \text{CaptureList.onClick.entry} \rightarrow \text{CaptureList.onClick.exit} \rightarrow \text{CamAct$3.onPictureTaken.entry} \rightarrow \text{CamAct$3.onPictureTaken.exit} \rightarrow \text{CaptureList.onTab.entry} \)

4. \( \text{CaptureList.onClick.entry} \rightarrow \text{CaptureList.onClick.exit} \rightarrow \text{CaptureList.onTab.entry} \rightarrow \text{CaptureList.onTab.exit} \rightarrow \text{CamAct$3.onPictureTaken.entry} \)

Note that we apply these steps for each API\_ASYNC in the CCFA. For example, if there
is another API\_ASYNC \( x \) that is invoked after \( \text{CamAct$3.onPictureTaken} \) in the API method
\( \text{cam.takePicture} \), we generate all the interleavings among the callbacks \( x \), \( \text{CaptureList.onClick} \)
and \( \text{CaptureList.onTab} \). Due to the timing issue, \( \text{CaptureList.onClick} \) and \( \text{CaptureList.onTab} \) may
be put in the event queue before or after \( x \) is put in the queue.

To measure the coverage for C3, we consider traces that involve API\_ASYNC and event call-
backs. Similar to measuring C1, for a trace \( T \), we do some filtering and exclude out all API\_SYNC
callbacks in the trace, resulting in $T_{C3} \subseteq T$. We say a sequence $s \in S_{C3}$ is covered if $T_{C3}$ contains $s$. Let $C_{C3}$ be the set of sequences covered by $T_{C3}$. The coverage C3 for the test is computed by $|C_{C3}|/|S_{C3}|$.

4.4 Evaluation

The goal of our evaluation is to empirically show that our coverage criteria can help testing quickly find certain types of bugs that statement and GUI-based coverage criteria are difficult, slow and sometimes impossible to trigger. Specifically, we aim to answer the following research questions:

RQ1: Can our coverage criteria, C1, C2 and C3, guide automatic input generation tools to find bugs in Android apps?

RQ2: Can testing based on C1, C2 and C3 find more bugs than testing based on GUI-based coverage?

RQ3: Can testing based on C1, C2 and C3 find bugs faster than testing based on GUI-based coverage?

4.4.1 Implementation and Experimental Setup

We also built a tool to instrument Android apps to collect traces using Soot [61]. Our tool uses the Jimple intermediate representation to insert instrumentation at the entry and exit points of callbacks and generates a signed APK for testing. When running the instrumented apps, we used logcat \(^2\) to collect callback traces from the phone logs. To calculate the coverage of C1, C2 and C3, we implemented the techniques described in Section 4.3. As a comparison, we also calculated the statement and GUI-based coverage. We collected statement coverage by instrumenting apps using Jacoco [1]. We calculated GUI-based coverage by identifying two consecutive GUI events on CCFAs and determining if they are in the trace.

\(^2\)https://developer.android.com/studio/command-line/logcat.html
We randomly selected 11 apps from the open source repository F-droid [39] as the benchmark. We used apps that do not require special inputs from the user such as a username and a password. This approach is used in other studies such as [45]. Table 4.3 presents the apps we selected with their categories and the total lines of source code (see Column SLOC).

We test the Android apps using the state-of-the-art tool Monkey [29]. Monkey [29] randomly explore the GUI events to test. To obtain the test cases that likely satisfy our criteria, we applied manual testing and use the sequences generated for C1, C2 and C3 to guide how we test the app. As an example of triggering C1, we move the phone to generate sensing events to exercise the callback sequences related to the sensing events [27]. Similarly, if there is a sequence of callbacks for C3 involving two different events and an API ASYNC, we trigger these two events with different timings to generate potential different interleavings between the three callbacks. We run each app from 15 to 30 minutes depending on the size of the app when we believe the potential callback sequences are all explored.

All the apps in our benchmark were run on a Motorola Nexus 6 with Android 7.1. To test the benchmarks with Monkey, we adopted the settings of the previous studies that use these tools [45, 58, 22, 62]. We use the default distribution of events and run each benchmark for 3 hours (see [62]).

<table>
<thead>
<tr>
<th>App</th>
<th>Category</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location Share</td>
<td>Navigation</td>
<td>384</td>
</tr>
<tr>
<td>Calculator</td>
<td>Tools</td>
<td>629</td>
</tr>
<tr>
<td>Pushup Buddy</td>
<td>Health &amp; Fitness</td>
<td>965</td>
</tr>
<tr>
<td>SdbViewer</td>
<td>Tools</td>
<td>1273</td>
</tr>
<tr>
<td>Cache Cleaner</td>
<td>Tools</td>
<td>1493</td>
</tr>
<tr>
<td>Movie DB</td>
<td>Entertainment</td>
<td>4727</td>
</tr>
<tr>
<td>BART Runner</td>
<td>Navigation</td>
<td>6124</td>
</tr>
<tr>
<td>Open Sudoku</td>
<td>Games</td>
<td>6440</td>
</tr>
<tr>
<td>Mileage</td>
<td>Finance</td>
<td>9931</td>
</tr>
<tr>
<td>Pedometer</td>
<td>Health &amp; Fitness</td>
<td>13502</td>
</tr>
<tr>
<td>Chanu</td>
<td>Social</td>
<td>45856</td>
</tr>
</tbody>
</table>
To answer RQ1, we measure how the coverage of C1, C2 and C3 increased before each bug (unique crash) is found. If we see the bugs often co-occur with the increase of the coverage, it suggests that testing guided by C1, C2 and C3 (and thus the increase of C1, C2 and C3) is useful for finding bugs. We measure the coverage of C1, C2 and C3 every 5 minutes during each test. After every 5 minutes, we compare how much the coverage increased from the previous measure. We then select each of the intervals when each bug was triggered for the first time and determine whether C1, C2 or C3 increased.

To answer RQ2, we first measure what is the coverage of C1, C2 and C3 achieved by existing testing tools. The most effective testing tools like Monkey [29] focus on generating GUI events. We want to demonstrate that these tools cannot achieve a high coverage of C1, C2 and C3, and thus testing based on C1, C2 and C3 potentially exercises more behavior of apps and find more bugs. We used manual testing to simulate the testing guided by C1, C2 and C3. We compared the coverage of C1, C2 and C3 achieved by Monkey with manual testing. We also compared the bugs found by Monkey with manual testing.

To answer RQ3, we compared the time takes to trigger the unique crashes using manual testing and existing tools. In the following three sections, we present our results for the three research questions.

### 4.4.2 Results for RQ1

In Table 4.4, we list each test criterion in the first column. Under Bugs, we report for all the 5-min periods where we observe the increase of the test coverage, what are the number of bugs first triggered (a bug may trigger multiple times in testing). In our study, we found a total of 28 bugs. C1 is most effective for guiding testing, and we found 82% of the bugs (23 bugs) when testing increases the coverage for C1, 18% more (5 more bugs) than the GUI-based criterion, and 44% more (13 more bugs) than the statement coverage. We found C3 is also very useful for finding bugs, as it found new bugs that C1 and GUI-based coverage cannot find. We manually analyzed the root causes of all the bugs and confirmed there are 8 race condition bugs. The occurrences of
(a) Coverage over time for Pedometer using Monkey

(b) Coverage over time for Movie DB using manual testing

Figure 4.5: Coverage over time for Pedometer and Movie DB
Table 4.4: The bugs found and their correlations with the increases of the test criteria

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>23 (82%)</td>
</tr>
<tr>
<td>C2</td>
<td>5 (17%)</td>
</tr>
<tr>
<td>C3</td>
<td>14 (50%)</td>
</tr>
<tr>
<td>GUI</td>
<td>18 (64%)</td>
</tr>
<tr>
<td>Statement</td>
<td>10 (38%)</td>
</tr>
</tbody>
</table>

these bugs are all correlated with the increases of C3. Whereas, GUI coverage is correlated with 3 of the 8 bugs and statement coverage is correlated with 2 of these bugs. Therefore, C3 is important for guiding testing to exercise the concurrency behavior of Android apps. If we consider C1 and C3 together, we found that the occurrences of 27/28 bugs are correlated with the increase of either C1 or C3.

In Figures 4.5a and 4.5b, we present two examples and show how the coverage for each criterion increased over time and the bugs found (light blue and red dots represent two unique crashes) for the apps Pedometer and Movie DB respectively. We observe that within the 5-minute periods of the occurrences of the bugs in Pedometer (in Figure 4.5a, see the sections of 20-25 minutes, and 175-180 minutes), the only coverage criterion increased is C3. Especially for the period of 175-180 minutes, the coverage of GUI and statements has not been updated for about 35 minutes. The developers who use the two criteria may already stop testing and miss the bug. Similarly for Movie DB (Figure 4.5b), we observe that after 10-min testing, only the coverage of C1 and C3 is updating. Thus, most likely, callback sequences of C1 and C3 had triggered the second bug for Movie DB.

4.4.3 Results for RQ2

In Table 4.5, we list the number of bugs found using different testing approaches. Our results show that Monkey and manual testing found a total of 8 common bugs. Although Monkey runs for hours and manual testing runs in minutes, by targeting C1 C2 C3, manual testing is able to find 7 bugs that cannot be found by Monkey. Monkey found 5 bugs missed by manual testing. Our inspection shows that 2 of these bugs are related to an Out of Memory exception and 2 are related to
Table 4.5: Unique crashes found by both tests, by just manual testing, or by just Monkey App

<table>
<thead>
<tr>
<th>App</th>
<th>Both</th>
<th>Manual</th>
<th>Monkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculator</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Movie DB</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>BART Runner</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>OpenSudoku</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Mileage</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pedometer</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>chanu</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>8</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

the event handlers that react to network connectivity events using the API `ConnectivityManager.registerNetworkCallback()`. These types of bugs require to execute the apps for a long time, which would be difficult for manual testing. We foreseen that an automatic input generation tool targeting C1, C2 and C3 can find these bugs that manual testing based on C1, C2 and C3 misses.

We also compare the coverage achieved by manual testing and Monkey for C1, C2 and C3. We want to investigate that whether the black box testing tool like Monkey can achieve C1, C2 and C3 if we run it enough time. As shown in Table 4.6, for 9 out of 11 apps, manual testing have improved the coverage of C1, C2 and C3 or at least achieve the same coverage compared to Monkey. The apps Location Share and Pushup Buddy have reported the biggest improvement for C1, as the manual testing is able to trigger the external events such as GPS location updates and sensor events while Monkey cannot test all sequences involving callbacks related to these events. For the apps such as BARTRunner and chanu, Monkey was able to detect more bugs and achieve a better coverage, as these apps mostly consist of the GUI callbacks, which Monkey targets.

4.4.4 Results for RQ3

We compare the time used by Monkey and manual testing to trigger each bug for the first time. Since Monkey takes a long time to trigger the bugs that manual testing did not find, we focus on comparing the bugs that were commonly discovered. See the row `Common bugs` in Table 4.7. We report the average, minimum and maximum time used in seconds. We observe that by targeting
Table 4.6: Coverage of C1, C2 and C3 using manual testing and Monkey

<table>
<thead>
<tr>
<th>App</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location Share</td>
<td>51.30</td>
<td>-</td>
<td>-</td>
<td>33.77</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Calculator</td>
<td>42.14</td>
<td>-</td>
<td>-</td>
<td>68.14</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Pushup Buddy</td>
<td>42.64</td>
<td>13.11</td>
<td>55.31</td>
<td>35.29</td>
<td>6.55</td>
<td>31.70</td>
</tr>
<tr>
<td>SdbViewer</td>
<td>44.15</td>
<td>-</td>
<td>59.13</td>
<td>38.29</td>
<td>-</td>
<td>58.62</td>
</tr>
<tr>
<td>Cache Cleaner</td>
<td>44.00</td>
<td>60.00</td>
<td>54.12</td>
<td>44.00</td>
<td>60.00</td>
<td>53.57</td>
</tr>
<tr>
<td>Movie DB</td>
<td>44.17</td>
<td>27.27</td>
<td>79.22</td>
<td>36.81</td>
<td>27.27</td>
<td>75.78</td>
</tr>
<tr>
<td>BARTRunner</td>
<td>32.68</td>
<td>-</td>
<td>31.45</td>
<td>27.35</td>
<td>-</td>
<td>51.58</td>
</tr>
<tr>
<td>Open Sudoku</td>
<td>45.78</td>
<td>25.45</td>
<td>76.58</td>
<td>24.64</td>
<td>7.27</td>
<td>60.00</td>
</tr>
<tr>
<td>Mileage</td>
<td>25.09</td>
<td>23.25</td>
<td>43.24</td>
<td>23.58</td>
<td>34.88</td>
<td>21.81</td>
</tr>
<tr>
<td>Pedometer</td>
<td>74.56</td>
<td>31.17</td>
<td>85.67</td>
<td>64.15</td>
<td>22.22</td>
<td>78.11</td>
</tr>
<tr>
<td>chanu</td>
<td>30.21</td>
<td>27.18</td>
<td>40.57</td>
<td>35.41</td>
<td>28.64</td>
<td>48.18</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>43.38</td>
<td>25.93</td>
<td>58.36</td>
<td>38.67</td>
<td>23.35</td>
<td>53.26</td>
</tr>
</tbody>
</table>

Table 4.7: Compare the time used to trigger the bugs

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Monkey</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg. (s)</td>
<td>Min (s)</td>
</tr>
<tr>
<td></td>
<td>327</td>
<td>4</td>
</tr>
</tbody>
</table>

C1, C2 and C3, manual testing is 32 times faster than Monkey on average. We observed that for bugs in apps such as Pedometer and chanu, Monkey took more than an hour to detect such bugs. The results show the C1, C2, C3 not only help testing find bugs that other testing tools cannot find, but even for the bugs the other tool can find, our testing criteria can help find bugs faster.

### 4.5 Summary

This chapter introduced 3 coverage criteria based on callback sequences, named C1, C2, and C3. C1 is designed to cover callback sequences from external events, including GUI, sensing and other types of events, C2 is designed to cover the different behaviors from callbacks invoked synchronously in API methods, and C3 is designed to cover the concurrent behavior between external events and callbacks invoked asynchronously in API methods.
Our evaluation results show that our coverage criteria are more effective in guiding tests to trigger more bugs than statement and GUI event-based coverage. We found that for 27 out of the 28 bugs detected in our studies, the behaviors to trigger the bugs are correlated with the increase of coverage of either C1 or C3. Moreover, we found that manual testing guided by our criteria is more effective and faster on detecting bugs than automatic input generation tools.
CHAPTER 5. RELATED WORK

In this chapter, we provide a discussion of related work. We present specific areas including the generation of pre-computed summaries for libraries, static and dynamic analyses of event-driven and framework-based applications, representations for control flow of event-driven systems, development coverage criteria and testing tools for event-driven systems and testing of concurrent systems.

**Pre-computed Summaries** EdgeMiner [18] is the closest work to our approach. Their goal is to map Android API methods to callbacks invoked in the methods. However, they do not sequence the callbacks. Our PCS technique identified more fine-grained information, including the order of callbacks and the conditions under which the callbacks will be invoked. This helps construct apps’ inter-callback ICFGs and also excludes infeasible paths. Pre-computed summaries to solve other types of program analysis problems have been defined. Clapp et al. [23] mine information flow specifications for the Android API methods. Their specifications identify how values from the app can be tainted in the Android API methods. Arzt et al. [7] developed a static analysis technique to generate data flow summaries from libraries to use in solving taint analysis problems. However, neither of these two works include any control flow information about callbacks. Rountev et al. [52, 54] extended the dataflow summaries generated by an interprocedural finite distributive subset (IFDS) and interprocedural distributive environment (IDE) algorithms to handle callback call sites in the libraries. The summaries are split before and after a callback call site and merged when the target callback method becomes available during client analysis. Ali et al. [3, 4] analyzed the application code using a single summary node to represent all library methods with a less conservative assumption (*separate compilation assumption*). This approach cannot be directly applied to Android apps as the app depends almost on its entire executions on the Android framework, invoking a great number of callbacks and framework method calls.
Static Analysis for Control Flow of Callbacks. Some analyses have focused on soundness and conservatively assume events can happen at any time. TAJS [33, 6] orders `onLoad` events but assumes any order in the rest of events in Javascript applications. Similarly, Liang et al. [38] compute all permutations between callbacks defined in Android apps. These approaches can be very imprecise since they do not consider any constraint on the ordering of callbacks imposed by the framework (e.g., callback `onStart` is always executed before callback `onResume` in activities).

Trying to use knowledge from the semantics of the system, several studies [48, 8, 36, 68, 31] have use lifecycle of components in Android apps to define inter-callback control flow. However, these approaches just focus on components’ callbacks and some GUI events and can be incomplete for the rest of the Android API [63]. Besides control flow constraints offered by the lifecycle of components, Blackshear et al. [12] use data dependencies between callbacks to identify more ordering constraints between callbacks.

These approaches offer a limited view of the constraints enforced by the system. As it is shown in [49], API calls can also have an impact on the control flow of callbacks, and but none of these approaches handle them systematically. Our representation includes control flow constraints on the ordering of callbacks enforced in external events such as GUI events as well as the API calls.

Dynamic Analysis for Control Flow of Callbacks. Most of the dynamic analyses on control flow of mobile event-driven systems focus on race detection problems. Maiya et al. [43] and Hsiao et al. [32] instrument various components of the Android API, including event-queue and memory operations to identify use-after-free race conditions. They define a causality model to reason about the possible ordering of callbacks. Our representation specifies asynchronous callbacks invoked through the event-queue by annotating the call sites and messages, but our inter-callback data flow analysis on the CCFAs does not model the impact of the event queue. We consider that our representation can be used together with the asynchronous semantic rules, similar to the rules defined in [43], to more accurately analyze asynchronous callbacks.

Control Flow Representations for Event-Driven Systems. Our approach of specifying asynchronous callbacks in CCFAs and especially, annotating transitions with the triggering events
and API calls is similar to the *emit* annotation on the *event-based call graph* [42], a representation that specifies the scheduling of event listeners in web applications. In addition, Dietrich et al. [28] construct a global control flow graph (GCFG) to handle RTOS semantics, including multiple kernel invocations in embedded systems. For mobile apps, Yang et al. [66, 67] created a graph representation (WTG) for GUI events and Activities’ stack semantics in Android apps. We showed that our representation can integrate all the control flow constraints found in WTGs as well as the callbacks invoked in API calls. We thus achieved higher callback coverage.

**Coverage Criteria for Event-Driven Systems.** Memon et al. [46] developed a family of coverage criteria for testing event-driven GUI applications. Their technique introduces a black-box model for event sequences to test permutations of events in GUI applications. For Android, most of the work on testing focus on the generation GUI events [5, 45, 20, 58, 41, 29] following similar techniques developed in [46]. According to a survey presented by Kong et al. [34], almost half (45.6%) the work related to testing of Android apps, focus on GUI testing.

Amalfitano et al. [5] presents the tool *AndroidRipper* which instead of dynamically generating a GUI model (called ripping) and then generate test input, AndroidRipper systematically test the while doing the ripping process. Mao et al. [45] developed SAPIENZ which combines random fuzzing and search-based exploration to maximize statement coverage. Su et al. [58] introduces *Stoat*, a tool that generates a weighted black-box GUI model dynamically, and uses a sampling technique to mutate the model in order to increase the result of an objective function. The objective function considers model coverage, statement coverage and model diversity (how did the GUI model change).

All these tools focus on some way to increase statement coverage or the coverage of a black-box GUI model. In this thesis, we focus on generating new coverage criteria and show that the GUI models and statement coverage lack important information that needs to be tested.

**Static Models for Testing.** In this thesis, we use a static model, CCFA, to generate callback sequences for C1, C2, and C3. Similarly, Azim and Neamtiu [9] implemented a technique (in the tool $A^3E$) that generates a control flow graph that contains legal transitions between Activities.
They also developed a targeted exploration technique to guide tests to improve coverage on their control flow graph. Yang et al. [66] developed Windows Transition Graphs (WTGs) to model sequences windows and GUI events. They then generate GUI events by traversing their WTGs. The WTGs has been also used for testing resource leaks [69]. Both of these approaches focus just on GUI behaviors. The CCFA covers GUI behaviors and also includes other external events (such as camera or sensors) and callbacks invoked in API methods. Neither $A^3E$ and WTGs cover these behaviors.

**Testing for Concurrency.** For testing concurrent systems, similar techniques to our work have been used on different systems. Deng et al. [25] and Choudhary et al. [21] used a pair of concurrent functions as a coverage metric for testing of C/C++ applications and thread-safe Java classes respectively. The former work uses the coverage metric for selection of pre-defined inputs whereas the latter use the metric for input generation. Similarly, Tasharofi et al. [59] developed different coverage criterion based on pairs of concurrent operations for actor programs. This related work shows that pairs of functions are an effective metric as coverage criteria for testing concurrent systems.

To help detect concurrency issues in Android, most of the related work used happen-before relation on dynamic traces. Hsiao et al. [32] and Maiya et al. [43] both developed concurrent models and happen-before relation for Android to detect race conditions between callbacks. [11] developed new techniques for scaling the inference of happens-before relations. All these techniques depend on dynamic traces generated from testing. Contrary, our coverage criterion C3 is generated from the CCFA which is a model for all the app and do not depend on testing to generate all possible sequences of concurrent callbacks. Li et al. [37] present a similar technique to ours to detect concurrency bugs between GUI callback listeners. They detect conflicted GUI callbacks and generate input events to test the interactions of these callbacks. In our work, coverage criterion C3 focus mostly on the conflicts that involve asynchronous callbacks invoked in API methods.
CHAPTER 6. CONCLUSIONS AND FUTURE WORK

The increase of popularity of smartphones and tablets have brought thousands of developers to publish apps in mobile markets. This creates a fierce competition when you can find several apps serving the same purposes. Given the complexity of developing apps, the need for new tools to verify the quality of apps has increased over the last years. However, the event-driven, framework-based nature mobile apps generate several challenges for applying previous program analysis and testing techniques.

This thesis identified one of the fundamental problems of analyzing event-driven and framework-based mobile apps which is the sequencing or ordering of callback methods invoked from external events (e.g. GUI events) and framework calls. We presented a representation, Predicate Callback Summary (PCS), and implemented tools to summarize control flow information from libraries. This representation allows us to identify the control flow that can be executed when an app makes API calls. Then, we integrated PCSs with a GUI model and external events models into a general model, CCFAs, to support control flow from external events and API calls. With a general model of execution for mobile apps, we implemented two applications and showed improvements over state-of-the-art tools.

Lastly, we use the CCFA to design new coverage criteria based on callback sequences. The key idea of our work is that a coverage criterion based on callback sequences can identify behavior that is not present in other coverage criteria such as statement coverage of GUI-based event sequences. We presented two studies to understand what callbacks are important in detecting bugs and designed 3 coverage criteria based on the results of the studies. The results of our evaluation showed that our coverage criteria are better suited to guide test input generation for Android apps than other coverage criteria such as statement and GUI event-based coverage.
6.1 Future Work

We foresee different avenues to extend our work presented in this thesis. First, the applications of CCFA to new mobile-specific problems. We also see an avenue of applying CCFAs on different domains. Second, as the next step of our coverage criteria defined in Chapter 4, we see an avenue to develop an automatic input generation tool or augment current tools using our coverage criteria to guide the tests. Last, we pretend to extend CCFA to handle more components of the Android framework such as *Fragments*.

6.1.1 Use of CCFA for different Analyses and Domains

We foresee that CCFAs may be used directly to develop new analysis for mobile apps. We also consider exploring the applications of CCFAs for other event-driven, framework-based systems. CCFA is a general model for event-driven systems that can enable a whole plethora of analyses once a system is modeled as a CCFA.

6.1.2 Coverage Guided Input Generation for Mobile Apps

Current tools use statement coverage as metric to decide whether tests are effective or not. For future work, we plan to design new input generation tools or augment current ones based on our criteria.

6.1.3 Extension of CCFAs

As we mentioned, our current algorithms to build CCFAs just handle GUI external events. There are several external events we are planning to include more events besides the current events we handle. Besides external events, there are important constructs from the Android framework that CCFA still does not model.
BIBLIOGRAPHY


Programming Language Design and Implementation, PLDI ’14, pages 259–269, New York, NY, USA. ACM.


