Specifying Callback Control Flow of Mobile Apps Using Finite Automata

Danilo Dominguez Perez and Wei Le

Abstract—Given the event-driven and framework-based architecture of Android apps, finding the ordering of callbacks executed by the framework remains a problem that affects every tool that requires inter-callback reasoning. Previous work has focused on the ordering of callbacks related to the Android components and GUI events. But the execution of callbacks can also come from direct calls of the framework (API calls). This paper defines a novel program representation, called *Callback Control Flow Automata (CCFA)*, that specifies the control flow of callbacks invoked via a variety of sources. 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 *Predicate Callback Summaries (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. Thus, callback control flow graphs and inter-callback analysis are indeed important. Although this paper mainly uses Android, we believe that CCFAs can be useful for modeling control flow of callbacks for other event-driven, framework-based systems.

Index Terms—event-driven, framework-based, mobile, program analysis, information flow

1 Introduction

With more than 1.4 billion Android devices sold [1] and over 2.5 million apps in the official Android app's market [2], Android has become one of the leading platforms for smartphone devices. However, the complex event-driven, framework-based architecture that developers use to implement apps imposes new challenges on the quality of Android apps, creating new kinds of code smells and bugs [3], [4], [5], [6]. Due to the impact of the Android apps on the market and the complexity of developing apps, there is a need for programming tools for analyzing Android apps. One of the foundational challenges for developing these tools is the sequencing or ordering of callback methods. 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 [7].

Previous tools have used manual models to obtain sequences of callbacks. For instance, Pathak et al. [8] and Arzt et al. [9] used the lifecycle of components to represent control flow between callbacks. This approach can miss sequences of callbacks invoked in API methods such as `AsyncTask.execute` and `LoaderManager.initLoader`. Other approaches modeled a subset of callbacks, specifically the callbacks invoked from external events including GUI [10], [11], [12] and sensors' events [13]. Another approach automatically summarize the Android framework to identify the control flow of callbacks implemented in the Android API methods [14]. Nevertheless, there is not a representation that integrates different sources of changes of control, such as external events and API methods, and be directly usable by analysis and testing tools.

In this paper, we propose 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) [15], 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 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.

In the 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 alphabet of the CCFA is composed by the names of callbacks implemented in the app. Specifically, we use $callbackname_{entry}$ and $callbackname_{exit}$ as input symbols so we can specify the case where one callback is invoked during the execution of the other (see Type 3 and Type 4 of control flow discussed above). 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) [10] to

• *Danilo Dominguez Perez and Wei Le are with the Department of Computer Science, Iowa State University, Ames, Iowa, 50011. E-mail: {danilo0,weile}@iastate.edu*

obtain the control flow of callbacks related to the windows (activities, dialogs and menus) and GUI events, and also, integrates *Predicate Callback Summaries* (PCSs) [14] for callback sequences invoked in the Android API methods. These API methods include the callbacks of the Android components' lifecycles, which are modeled manually in the previous work. Our approach is able to automatically handle them. In addition, we successfully added the callbacks for 9 additional external events: GPS location updates, enabling and disabling of GPS services, sensor events, two camera external events (when the camera shutters and when the camera takes a picture), network notifications (through the `ConnectivityManager`), a web client and a web chrome client that receive notifications from a `WebView`. The models for these events are constructed manually, but we show that CCFA can be easily extended to specify external events beyond the window and GUI events.

In this paper, we show two different approaches of integrating CCFAs into 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 a program using the callback invocation paths provided by CCFAs to directly integrate CCFAs with *FLOWDROID* [9].

We implemented the construction and applications of CCFAs using *Soot* and *Spark*. The WTGs were computed using *GATOR* [16] and the summaries for Android API methods were provided by *Lithium* [14].

Our results show that the construction of CCFAs is very efficient, reporting 29 seconds per app on average. Compared to WTGs, CCFAs improved callback coverage 10.72% on average. In the best cases, we improved the app `heregps`'s coverage from 50% to 100%, and for the app `MyTracks`, 78 more callbacks were integrated, improving the callback coverage 15.30%. Applying CCFAs, our tool found 33 more information leaks than *FLOWDROID* for 55 apps we ran. We also experimented on 60 apps from *DroidBench* that contain inter-callback security leaks, and our tool reports 96.00% precision and 84.21% recall. Importantly, we found that on average 22.76% of the source and sink pairs have the sources and sinks located in different callbacks. These results validated the hypothesis that the inter-callback analysis and callback control flow graphs are important for analyzing the global behaviors of the apps.

In summary, this paper makes the following contributions:

- We define the *Callback Control Flow Automata CCFA*, a representation to specify callback control flow for event-driven, framework-dependent applications such as Android apps,
- We design the algorithm to automatically compute CCFAs using apps and framework source code,
- We design inter-callback dataflow analysis to report information leak and source-sink pairs using CCFAs, and
- We implemented the tools for constructing and using the CCFAs and experimentally demonstrated that our algorithms are fast and are able to discover inter-callback properties that previous tools cannot find.

The rest of the paper is organized as follows. In Section 2, we present an overview of our work. In Sections 3 and 4, we present the definition of CCFAs and the algorithms for computing CCFAs. In Section 5 we show how to use CCFAs for inter-callback analysis. The evaluation of our work is presented in Section 6, followed by a discussion in Section 7 and the related work in Section 8. Finally, we present the conclusions and future work in Section 9.

2 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.

2.1 What is a CCFA

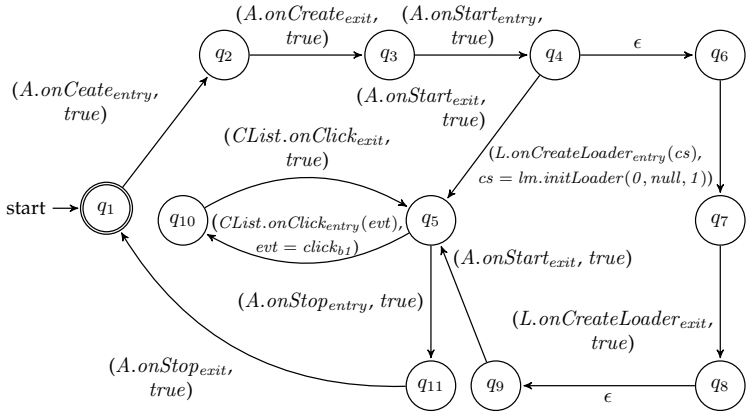
In Figure 1a, we show a simple Android app that contains one activity class called A. This activity implements the callbacks `onCreate` (line 3), `onStart` (line 7) and `onStop` (line 15). In the callback `onCreate` at line 5, the app sets an object of type `CList` as the event listener for the click event of the button `b1`. It also invokes the API method $lm.initLoader$ at line 12 to load data, where this API call uses the class `L` (lines 17-22) to create a loader.

Figure 1b shows the CCFA we constructed for Figure 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 ϵ (see the 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 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 trivial $true$ guard.

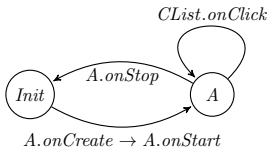
This representation specifies both synchronous and asynchronous invocations of callbacks and we are able to recognize them based on the transitions' parameters and guards. Sequential invocation of callbacks is specified using the guard `true` or the guard related to the cs parameter. As an example, the transition from state q_3 to state q_4 in Figure 1b indicates that the execution of `A.onStart` immediately follows `A.onCreate`. CCFAs can also specify

```
1  class A extends Activity {
2    Button b1;
3    void onCreate(Bundle b) {
4      b1 = (Button) findViewById(...);
5      b1.setOnClickListener(new CList());
6    }
7    void onStart() {
8      ...
9      if (*) {
10       L l = new L();
11       LoaderManager lm = getLoaderManager();
12       lm.initLoader(0, null, l);
13     }
14   }
15   void onStop() { ... }
16 }
17 class L implements LoaderCallbacks {
18   Loader onCreateLoader(int i,
19                         Bundle b) {
20     ...
21   }
22 }
23 class CList extends OnClickListener {
24   void onClick(View v) { ... }
25 }
```

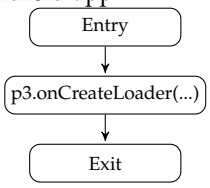
(a) Source code of a simple Android app



(b) CCFA for the Android app



(c) WTG for the Android app



(d) PCS for `initLoader`

Fig. 1: Source code, CCFA, WTG and PCS of an example app

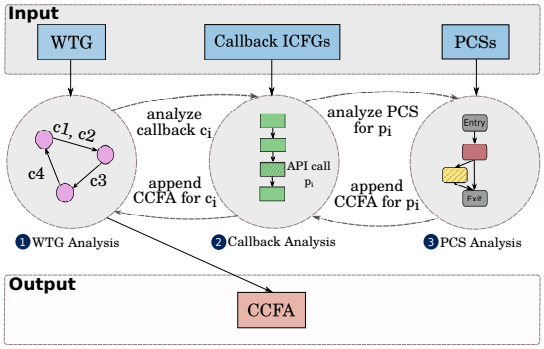


Fig. 2: An Overview of the Approach

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 1b). Callbacks that are parametrized with external events are executed asynchronously. For instance, for the app in Figure 1a, when a user clicks button `b1`, the framework puts a task into an event queue, which eventually executes the callback `CList.onClick`. In Figure 1b, the transition between q_5 and q_{10} specifies this control flow transfer.

2.2 How to Compute a CCFA

In Figure 2, we show that our approach to computing CCFAs takes a Windows Transition Graph (WTG) [10], the interprocedural control flow graphs (ICFGs) of callbacks in the app, and the *Predicate Callback Summaries* (PCSs) of the Android framework [14] 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.

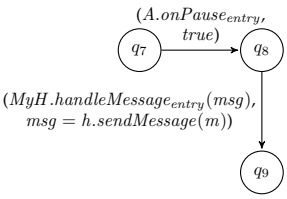
Performing on the three inputs, our approach consists of three major analyses: *WTG Analysis*, *Callback Analysis*, and *PCS Analysis*. As shown in Figure 2, in the first step, the WTG Analysis traverses the WTG to obtain callback sequences located on the edges of WTG. For each callback, the WTG Analysis 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 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 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 1a, the callback `A.onStart` has two paths and only one of the paths invokes the API call $lm.initLoader$. Correspondingly, in Figure 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 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 1b are generated by the PCS Analysis using the PCS shown in Figure 1d.

```
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`

Fig. 3: Guard for `sendMessage`

3 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.

3.1 Definition

The CCFA is a representation based on the computation model of Extended Finite State Machines (EFSMs) [8].

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 $\{x_{entry} | \ x \in C\}$ to denote the entry points of the callbacks (the entry node of the ICFG of the callback), and $\{x_{exit} | \ x \in C\}$ for the exit points (the exit node of the ICFG). The input symbols are defined as: $X = \{x_{entry} \cup x_{exit} | \ x \in C\}$. 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 $X = \{A.onCreate_{entry}, \ A.onCreate_{exit}, \ A.onStart_{entry}, A.onStart_{exit}\}$. 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: evt, the external events, cs, the API call sites, and msg, the asynchronous messages.

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 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 1b is labeled with the input $Clist.onClick_{entry}$, the input parameter for this input is evt, and the guard that uses the input parameter is $evt = click_{b1}$, indicating that the event $click_{b1}$ 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 *back button* or *power button*, and the actions such as *rotation of the phone*.

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 cs with the input symbol $x_{entry}(cs)$ 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 api method. As a concrete example, in Figure 1b, the transition from states q_6 to q_7 is labeled with the input symbol $L.onCreateLoader_{entry}$ and its parameter cs, and the guard $cs = lm.initLoader(0, null, 1)$. The transition indicates that `L.onCreateLoader` is the first callback executed in the API call `initLoader`.

msg: There are two special Android API methods, `Handler.sendMessage` and `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 *runnable objects* respectively to the event queue. Correspondingly, when the message is dequeued, the framework execute callbacks `Handler.handleMessage` and `Runnable.run` respectively. In CCFAs, the transitions to the entry points of callbacks `Handler.handleMessage` (or the callback `Runnable.run`) are parametrized using the input 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 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 q_8 to state q_9 in Figure 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 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 in this paper, 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.

4 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 2 in Figure 2 and consist of three main procedures that call each other to generate a final CCFA. Specifically, Algorithm 1 represents *WTG Analysis* in Figure 2, Algorithm 2 represents *Callback Analysis*, and Algorithm 3 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 [10]. Let Win be the set of all windows (activities, menus or dialogs) defined in an app and C be the set of callbacks implemented in an app. A *WTG* is a directed graph $G = (Win, E, \lambda, \delta, \sigma)$ with nodes $w \in Win$ and edges $e \in E \subseteq Win \times Win$. 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 λ, δ and σ. The function $\lambda : E \rightarrow Event$ returns the event that triggered the transition, whereas the function $\delta : E \rightarrow (push, pop \times Win)^*$ returns the window stack operations. The most relevant to this work is the function $\sigma : E \rightarrow ((Win \cup View) \times C)^*$ that returns the sequence of callbacks potentially executed during the transition. Figure 1c presents an example of WTG.

A *Predicate Callback Summary (PCS)* summarizes the control flow of callbacks invoked in the Android API methods [14]. 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 *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 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 1: Compute a CCFA for an app P

```
   input : WTG (model for windows and GUI), P (App
           code), Z (PCSs)
   output: CCFA (Callback Control Flow Automata)
1  init ← getState(entry(WTG)); Seen
     ← {(entry(WTG), init)};
2  Worklist ← {(entry(WTG), init)}; CCFA ← (init, {}, {});
3  while Worklist ≠ ∅ do
4      (node, s0) ← remove(Worklist);
5      s1 ←getState(node);
6      foreach edge ∈ succ(node, WTG) do
7          g ← createEvtGuard(edge);
8          while c = getNextCallback(edge) do
9              if c is last then s2 ← getState(edge.tgt);
10             else s2 ← getState(null);
11             CCFA = extend(CCFA, c, s1, s2, s0, g);
12             g ← true; s1 ← getState(null); s0 ← s2;
13         end
14         if (edge.tgt, s0) ∉ Seen then
15             add(edge.tgt, s0) to Seen and Worklist
16         end
17     end
18     if node is exit then  CCFA.finalStates = CCFA.finalStates
         ∪ {s0};
19 end
20 procedure extend(CCFA, c, s1, s2, s0, g) {
21     add transition (s0, c_entry, g, s1) to CCFA;
22     Fc ← analyzeCallback(c, s1);
23     if Fc is not empty then
24         add all transitions from Fc to CCFA;
25         foreach sf ∈ Fc.finalStates do
26             add transition (sf, c_exit, true, s2) to CCFA;
27         end
28     end
29     else add transition (s1, c_exit, true, s2) to CCFA;
30     return CCFA;
31 }
```

Shown in Algorithm 1, 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 s_0, 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 s_1 for the WTG node using `getState`, and at lines 9 and 10, we create

the state s_2. 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 s_0, s_1 and s_2, the callback c, and the guard g as inputs, and creates two transitions, one from s_0 to s_1 (see line 21), and the other from s_1 and s_2 (see line 29). It also analyzes the callback c, resolves any API calls in the callback, and generates a CCFA F_c for the callback c (see line 22). F_c takes s_1 as its initial state. The details of `analyzeCallback` are given in Algorithm 2. 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, s_0, is marked as a final state.

Algorithm 2 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 n_0. 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 s_0 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 `sendMesssage` 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 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 3a has the type `MyH`. Finally, at line 8, the same `extend` function (see lines 20–31 in Algorithm 1) 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 3) 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 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 the 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. [17] to handle loops.

ALGORITHM 2: Function analyzeCallback used in Algorithm 1

```
   input : c (Callback), s (initial state)
   output: F (CCFA for callback c with initial state s)
1  G ← buildICFG(P, c); n0 ← entryNode(G);
2  Worklist ← {(n0, s)}; Seen ← {(n0, s)}; F ← (s, {}, {});
3  while Worklist ≠ ∅ do
4      (n, s0) ← remove(Worklist);
5      if n is a call to sendMessage or post then
6          c ← getCallback(n); g
             ←createMsgGuard(n);
7          s1 ← getState(null); s2 ← getState(null);
8          F ← extend(F, c, s1, s2, s0, g); s0 ← s2;
9      end
10     else if n is an API call then
11         FPCS ← analyzePCS(n);
12         if FPCS is not empty then
13             add all transitions from FPCS to F;
14             add transition(s0, ϵ, true, FPCS.initialState) to
                F;
15             s0 ← getState(null);
16             foreach sf ∈ FPCS.finalStates do
17                 add (sf, ϵ, true, s0) to F;
18             end
19         end
20     end
21     else if n is an exit node then F.finalStates = F.finalStates
        ∪ {s0};
22     foreach succ ∈ succs(n, G) do
23         if (succ, s0) ∉ Seen then
24             add (succ, s0) to Worklist and Seen
25         end
26     end
27 end
28 return F;
```

In Algorithm 3, 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 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 [18]). The algorithm uses a worklist to traverse the PCS at lines 6 to 18.

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 $init$, the guard is g_{cs}; otherwise the guard is $true$ (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 `findCallbacks` to obtain the possible implementations of callbacks of the current node n. This function first gets the possible types of the callback using $refs$ 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 $unknown$, 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 `propagatePCS` finds the successors of the current node n and add them to the worklist.

ALGORITHM 3: Function analyzePCS used in Algorithm 2

```
   input : stmt (API call site)
   output: F (CCFA for the API call)
1  PCS← getPCS(Z, stmt); g_cs
     ← createAPIGuard(stmt);
2  refs ← resolveReferences(stmt);
3  n_0 ← entryNode(PCS); init ← getState(null);
4  Worklist ← {(n_0, init)}; Seen ← {(n_0, init)};
5  F ← (init, {}, {});
6  while Worklist ≠ ∅ do
7      (n, s_0) ← remove(Worklist);
8      if n is callback node and P implements it then
9          s_1 ← getState(null); s_2 ← getState(null);
10         if s_0 = init then g = g_cs else g ← true;
11         foreach c ∈ findCallbacks(n, P, refs) do
12             F ← extend(F, c, s_1, s_2, s_0, g);
13         end
14         s_0 ← s_2;
15     end
16     else if n is exit node then F.finalStates = F.finalStates
         ∪{s_0};
17     propagatePCS(n, PCS, s_0, Worklist, Seen);
18 end
19 return F;
```

5 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 [19] 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 [9] 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 $true$.

5.1 First Approach

In Algorithm 4, we present the inter-callback dataflow analysis extended from [19]. 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.

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

ALGORITHM 4: Inter-callback Dataflow Analysis using CCFAs

```
   input : ICFGs (set of ICFGs for the callbacks in the
           app), L (semi-lattice), F, (CCFA for the app), x_0
           (an initial context)
   output: S[N] of L
1  foreach t ∈ transitions from initial state F.s_0 with guard true
     do
2      n_0 ← entry(ICFGs, t.i);
3      add (n_0, x_0, t.s_2, null, t.i) to Worklist; H[n_0, x_0] ← x_0
4  end
5  while Worklist ≠ ∅ do
6      remove (n, x, s, c, i) from Worklist; y ← H[n, x];
7      if n is an API call node then
8          C[n] ← C[n] ∪ {(c, x)};
9          foreach t ∈ transitions from s with guard cs = n or
             msg = n do
10             n_c ← entry(ICFGs, t.i);
               y_c ← f_c(y, n, t.i.o);
11             propagate(n_c, y, y_c, t.s_2, n, t.i);
12         end
13     end
14     else if n is an exit of callback then
15         y_r ← f_rc(y, x, c, i.o);
16         S_n ← successors of s with transition labeled with
             i_exit;
17         foreach t ∈ transitions from states of S_n with guard
             true do
18             if t.i is 'ϵ' then
19                 foreach (c_p, x_p) ∈ C[n] and s ∈ succ(c)
                     do
20                     propagate(s, x_p, y_r, t.s_2, c_p, t.i)
21                 end
22             end
23             else
24                 n_c ←entry(ICFGs, t.i);
25                 y_c ← f_c(y_r, c, t.i.o);
26                 propagate(n_c, y_r, y_c, t.s_2, c, t.i)
27             end
28         end
29     end
30     else ... // stays the same as the algorithm in [19]
31 end
```

(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 [19] 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 [19].

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 4 for a value flow analysis using a similar approach specified in [20]. 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 [20]), and the second pass queries the dataflow solution at the node to identify the source statements of each used value.

5.2 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 [7]. 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. [21]. 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 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.

6 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.

6.1 Implementation and Experimental Setup

We implemented our algorithms using Soot [18] and *Spark* [22] for pointer analysis and call graph construction. To generate the WTG, we extended GATOR [10] 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 [23]. The 75 real-world apps include 55 random selected apps from the F-droid repository [24] and 20 apps used in [10], 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 6.2). We then compare the callback coverage of CCFAs versus WTGs (Section 6.3). Next, we demonstrate the improvement of information flow analysis using CCFAs compared to the existing control flow model, the Lifecycle of the components (Section 6.4). Finally, we report the results of inter-callback value flow analysis using CCFAs (Section 6.5).

6.2 Performance

In Table 1, 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 [25], 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 4, 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, chanu 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.

TABLE 1: Performance of Computing CCFAs

App	Category	Stmt	CB	WTG	CCFA	T (s)
heregps	Navigation	239	8	4 (50.00%)	8 (100.00%)	1
ContactsList	Communication	574	22	10 (45.45%)	16 (72.73%)	1
SDScanner	Tools	1001	10	5 (50.00%)	6 (60.00%)	1
Obsqr	Tools	1437	18	5 (27.78%)	6 (31.58%)	1
CamTimer	Photography	1926	20	11 (55.00%)	13 (65.00%)	1
VuDroid	Books	2826	35	11 (31.43%)	16 (45.71%)	1
StandupTimer	Productivity	2835	49	31 (63.27%)	38 (77.55%)	1
Oscilloscope	Tools	3114	53	15 (28.30%)	25 (47.17%)	1
TippyTipper	Finance	3266	74	48 (64.86%)	49 (66.22%)	1
DroidLife	Games	3270	48	28 (58.33%)	34 (70.83%)	1
SuperGenPass	Tools	3377	54	17 (31.48%)	21 (38.89%)	1
BARIA	Tools	3543	80	14 (17.50%)	19 (23.75%)	1
DrupalEditor	Productivity	4269	69	49 (71.01%)	65 (94.20%)	2
HaRail	Navigation	4424	38	14 (36.84%)	15 (39.47%)	2
Reservator	Productivity	4499	62	37 (59.68%)	40 (64.52%)	1
DiveDroid	Sports	4617	23	11 (47.83%)	11 (47.83%)	1
OpenManager	Tools	4912	49	28 (57.14%)	34 (69.39%)	1
DoliDroid	Business	5513	51	30 (58.82%)	44 (86.27%)	1
APV	Productivity	6093	66	37 (56.06%)	45 (68.18%)	1
arXiv	Education	6200	51	40 (78.43%)	40 (78.43%)	3
Notepad	Productivity	6411	111	43 (38.74%)	60 (54.05%)	1
JustCraigslist	Shopping	6624	45	33 (73.33%)	41 (91.11%)	2
CarCast	Audio	9005	108	63 (58.33%)	64 (59.26%)	2
OpenSudoku	Games	7857	111	43 (38.74%)	61 (54.95%)	1
EPMobile	Medical	12337	181	140 (77.35%)	143 (79.01%)	3
Giggity	Events	16060	116	36 (31.03%)	61 (52.59%)	5
MPDroid	Video	20454	166	72 (43.37%)	84 (50.60%)	1
a2dp.Vol	Navigation	10172	156	55 (35.26%)	92 (58.97%)	3
SimpleLastfm	Audio	12035	98	36 (36.73%)	42 (42.86%)	3
BasketBuild	Tools	13523	25	11 (44.00%)	14 (56.00%)	26
Currency	Finance	14908	81	25 (30.86%)	34 (41.98%)	3
KeePassDroid	Tools	15386	115	77 (66.96%)	83 (72.17%)	13
MovieDB	Entertainment	16742	156	28 (17.95%)	53 (33.97%)	96
Mileage	Finance	16966	197	78 (39.59%)	97 (49.24%)	6
droidar	Games	20177	113	24 (21.24%)	26 (23.01%)	6
NPR	News	24637	130	38 (29.23%)	70 (53.84%)	9
RadioDroid	Audio	25953	175	15 (8.57%)	36 (20.57%)	4
BarcodeScanner	Tools	29687	76	41 (53.95%)	48 (63.16%)	7
Beem	Communication	31253	193	62 (32.12%)	105 (54.40%)	4
SipDroid	Communication	31981	110	40 (36.36%)	44 (40.00%)	115
Aard	Books	32316	90	25 (27.78%)	39 (43.33%)	24
VLC	Audio	34637	570	90 (15.79%)	107 (18.77%)	4
XBMC	Video	34668	336	155 (46.13%)	167 (49.70%)	5
NewsBlur	News	34871	314	70 (22.29%)	79 (25.16%)	666
Connectbot	Communication	36953	255	83 (32.55%)	110 (43.14%)	9
CallRecorder	Tools	43646	196	21 (10.71%)	31 (15.82%)	29
Flym	News	48162	212	25 (11.79%)	41 (19.34%)	3
Etar	Productivity	63443	592	49 (8.28%)	67 (11.32%)	34
Munch	News	65594	270	24 (8.89%)	24 (8.89%)	44
Domodroid	Tools	68243	266	83 (31.20%)	110 (41.35%)	43
Calculator	Tools	68468	448	27 (6.03%)	40 (8.93%)	35
MyTracks	Navigation	84586	510	99 (19.41%)	177 (34.71%)	44
chanu	Social	100039	535	94 (17.57%)	117 (21.87%)	89
k9	Communication	115219	738	114 (15.45%)	155 (21.00%)	48
astrid	Productivity	143043	1245	191 (15.34%)	237 (19.04%)	231
Summary (Avg.)		**25163**	**180**	**38.04%**	**48.76%**	**29**

6.3 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 1, 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.

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.

6.4 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 5. 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* [26] to enable *IccTA* [27].

Compared with FLOWDROID on real-world apps. In Table 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 [21]. 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.

Compared with ground truth given by DroidBench. In the second experiment, we selected 60 apps from *DroidBench* [23]. *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, 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.

TABLE 2: FLOWDROID on lifecycle model and on CCFA

Benchmark	Lifecycle		CCFA	
	TP	FP	TP	FP
a2dp.Vol	4	0	4	0
APV	1	0	1	0
arXiv	3	0	3	0
CarCast	2	0	2	0
Connectbot	4	0	0	0
DrupalEditor	3	0	3	0
Etar	36	0	36	0
Flym	2	14	0	0
Giggity	6	0	6	0
KeePassDroid	16	0	16	0
NewsBlur	4	0	4	0
Notepad	15	11	17	11
NPR	12	0	42	5
osmdroid	35	0	0	0
SDScanner	2	0	2	0
SimpleLastfmScrobbler	255	0	255	0
StandupTimer	2	0	2	0
SuperGenPass	3	0	3	0
Reservator	0	0	1	0
VLC	376	0	376	0
XBMC	276	0	276	0

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 re-

ported a false negative in which an external file is tainted with sensitive information and later is read to leak the information.

TABLE 3: Information Flow Analysis on DroidBench

Category	Apps	CCFA			FLOWDROID		
		TP	FP	FN	TP	FP	FN
Lifecycle	17	15	0	2	15	0	2
Callbacks	15	16	1	0	15	0	1
InterCommunication	16	8	0	6	7	0	7
AndroidSpecific	12	9	1	1	9	0	1

6.5 Value-Flow Analysis for Source-Sink Pairs

Here, we report our results of value-flow analysis described in Section 5. 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 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* [10] 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 [9]) or computed stubs [28] to summarize such dataflow facts.

TABLE 4: Global Source Sink Pairs

Benchmark	Total	Inter-callback	Inter-comp
heregps	163	50	0
ContactsList	99	51	0
SDScanner	640	224	0
Obsqr	262	109	0
CamTimer	1290	353	0
VuDroid	89	14	0
StandupTimer	1037	402	0
Oscilloscope	481	47	0
TippyTipper	537	116	0
DroidLife	1453	322	3
SuperGenPass	357	28	0
BARIA	923	378	43
DrupalEditor	409	78	5
HaRail	269	16	0
Reservator	2641	1076	21
DiveDroid	1670	158	33
OpenManager	2969	863	0
DoliDroid	2487	498	1
APV	3756	699	2
arXiv	1249	51	0
Notepad	1784	500	7
JustCraigslist	6699	4161	8
CarCast	3123	541	56
OpenSudoku	233	13	0
EPMobile	5037	1808	388
Giggity	4006	679	2
MPDroid	896	255	0
a2dp.Vol	2544	981	75
SimpleLastfmScrobbler	2971	588	107
BasketBuildDownloader	322	85	0
Currency	508	148	0
KeePassDroid	1018	183	122
MovieDb	4113	638	262
Mileage	7174	1765	6
droidar	4610	305	80
NPR	2422	517	62
RadioDroid	626	114	0
BarcodeScanner	9871	52	0
Beem	3235	340	0
SipDroid	16073	4295	3941
Aard	613	173	0
VLC	1466	224	0
XBMC	5691	1000	12
NewsBlur	13876	4517	3494
Connectbot	2373	573	58
CallRecorder	2480	494	21
Flym	1619	408	15
Etar	17544	2803	14
Munch	2177	315	13
Domodroid	12391	434	0
Calculator	5782	1155	0
MyTracks	5737	1138	421
chanu	13147	1207	266
k9	12558	2264	1261
astrid	27329	10959	3045
Summary (Avg.)	**4088**	**930 (22.76%)**	**251 (6.16%)**

7 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 2 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 4, 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 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 paper. We plan to implement such improvement in the future work.

8 Related Work

Static Analysis for Control Flow of Callbacks. Some analyses have focused on soundness and conservatively assume events can happen at any time. *TAJS* [29], [30] orders *onLoad* events but assume any order in the rest of events in Javascript applications. Similarly, Liang et al. [31] 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 [4], [5], [9], [27], [32], [33] 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 [7]. Besides control flow constraints offered by lifecycle of components, Blackshear et al. [34] 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 [14], 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. [35] and Hsiao et al. [36] 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 dataflow 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 [35], 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* [37], a representation that specifies the scheduling of event listeners in web applications. In addition, Dietrich et al. [38] construct a global control flow graph (GCFG) to handle RTOS semantics, including multiple kernel invocations in embedded systems. For mobile apps, Yang et al. [10], [39] 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.

9 Conclusions

This paper 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. We foresee that CCFAs may be used with tools such as *Java Path Finder* (JPF) to generate test inputs, and be useful to measure the test coverage of existing tools. As future work, we will explore more applications of CCFAs like detecting sensor leaks [40], and also consider exploring the applicabilities of CCFAs for other event-driven, framework-based systems such as `Node.js` [41].

Acknowledgments

Danilo Dominguez Perez was supported by IFARHU-SENACYT scholarships from the Government of Panama.

REFERENCES

[1] J. Callaham, "Google says there are 1.4 billion active Android devices worldwide," Sep. 2015. [Online]. Available: http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide

[2] Statista, "Number apps available in Google Play Store," Jun. 2017. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[3] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in Android applications," in *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*. IEEE, 2016, pp. 225–236.

[4] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 267–280.

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

[6] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1013–1024. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568229

[7] Y. Wang, H. Zhang, and A. Rountev, "On the unsoundness of static analysis for Android GUIs," in *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 2016, pp. 18–23.

[8] K.-T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *30th ACM/IEEE Design Automation Conference*, June 1993, pp. 86–91.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[10] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, and A. Rountev, "Static window transition graphs for Android," *Automated Software Engineering*, vol. 25, no. 4, pp. 833–873, Dec 2018. [Online]. Available: https://doi.org/10.1007/s10515-018-0237-6

[11] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261.

[12] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 641–660. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509549

[13] T. Griebe, M. Hesenius, and V. Gruhn, "Towards automated UI-tests for sensor-based mobile applications," in *Intelligent Software Methodologies, Tools and Techniques*, H. Fujita and G. Guizzi, Eds. Cham: Springer International Publishing, 2015, pp. 3–17.

[14] D. D. Perez and W. Le, "Generating predicate callback summaries for the Android framework," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2017, pp. 68–78.

[15] A. Petrenko, S. Boroday, and R. Groz, "Confirming configurations in EFSM testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 29–42, 2004.

[16] P. Analyses and S. T. P. R. G. at the Ohio State University, "Gator: Program analysis toolkit for Android," 2017. [Online]. Available: http://web.cse.ohio-state.edu/presto/software/gator/

[17] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 174–184. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273487

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-A java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[19] A. Rountev, B. G. Ryder, and W. Landi, "Data-flow analysis of program fragments," in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7. London, UK, UK: Springer-Verlag, 1999, pp. 235–252. [Online]. Available: http://dl.acm.org/citation.cfm?id=318773.318945

[20] A. De and D. D'Souza, "Scalable flow-sensitive pointer analysis for Java with strong updates," *ECOOP 2012–Object-Oriented Programming*, pp. 665–687, 2012.

[21] S. Blackshear, A. Gendreau, and B.-Y. E. Chang, "Droidel: A general approach to Android framework modeling," in *Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*. ACM, 2015, pp. 19–25.

[22] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using Spark," in *Compiler Construction*. Springer, 2003, pp. 153–169.

[23] S. S. E. at Paderborn University and T. Darmstadt, "Droidbench - benchmarks," 2016. [Online]. Available: https://blogs.uni-paderborn.de/sse/tools/droidbench/

[24] F.-D. Limited and Contributors, "F-droid," 2017. [Online]. Available: https://f-droid.org/

[25] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying Java bytecode for analyses and transformations," 1998.

[26] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 77–88. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818767

[27] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 280–291. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818791

[28] S. Arzt and E. Bodden, "Stubdroid: Automatic inference of precise data-flow summaries for the Android framework," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 725–735.

[29] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the HTML DOM and browser API in static analysis of JavaScript Web applications," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 59–69. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025125

[30] E. Andreasen and A. Møller, "Determinacy in static analysis for jQuery," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 17–31.

[31] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, P. Aldous, and D. Van Horn, "Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation," in *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, ser. SPSM '13. New York, NY, USA: ACM, 2013, pp. 21–32. [Online]. Available: http://doi.acm.org/10.1145/2516760.2516769

[32] Z. Yang and M. Yang, "Leakminer: Detect information leakage on Android with static taint analysis," in *Software Engineering (WCSE), 2012 Third World Congress on*. IEEE, 2012, pp. 101–104.

[33] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1329–1341. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660357

[34] S. Blackshear, B.-Y. E. Chang, and M. Sridharan, "Selective control-flow abstraction via jumping," in *ACM SIGPLAN Notices*, vol. 50, no. 10. ACM, 2015, pp. 163–182.

[35] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for Android applications," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*,

ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 316–325. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594311

[36] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race detection for event-driven mobile applications," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 326–336. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594330

[37] M. Madsen, F. Tip, and O. Lhoták, "Static analysis of event-driven Node.Js JavaScript applications," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 505–519. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814272

[38] C. Dietrich, M. Hoffmann, and D. Lohmann, "Cross-kernel control-flow–graph analysis for event-driven real-time systems," in *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM*, ser. LCTES'15. New York, NY, USA: ACM, 2015, pp. 6:1–6:10. [Online]. Available: http://doi.acm.org/10.1145/2670529.2754963

[39] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in Android applications," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 89–99.

[40] H. Wu, Y. Wang, and A. Rountev, "Sentinel: Generating GUI tests for Android sensor leaks," in *Proceedings of the 13th International Workshop on Automation of Software Test*, ser. AST '18. New York, NY, USA: ACM, 2018, pp. 27–33. [Online]. Available: http://doi.acm.org/10.1145/3194733.3194734

[41] N. Foundation, "Node.js," 2018. [Online]. Available: https://nodejs.org/en/