2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Static Control-Flow Analysis of User-Driven Callbacks in Android Applications

Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev
Ohio State University
Email: {yangs,yan,wuhaow,wang10,rountev}@cse.ohio-state.edu

***Abstract*—Android software presents many challenges for static program analysis. In this work we focus on the fundamental problem of static control-flow analysis. Traditional analyses cannot be directly applied to Android because the applications are framework-based and event-driven. We consider user-event-driven components and the related sequences of callbacks from the Android framework to the application code, both for lifecycle callbacks and for event handler callbacks.**

We propose a program representation that captures such callback sequences. This representation is built using context-sensitive static analysis of callback methods. The analysis performs graph reachability by traversing context-compatible interprocedural control-flow paths and identifying statements that may trigger callbacks, as well as paths that avoid such statements. We also develop a client analysis that builds a static model of the application's GUI. Experimental evaluation shows that this context-sensitive approach leads to substantial precision improvements, while having practical cost.

I. Introduction

In recent years the growth in the number of computing devices has been driven primarily by smartphones and tablets. For such devices, Android is the dominating platform. A recent report estimates that more than 1.3 billion Android devices will be shipped in 2015, and this number will be larger than the combined number of all shipped Windows/iOS/MacOS PCs, notebooks, tables, and mobile phones [1].

The widespread use of Android software presents numerous challenges for software engineering researchers. One such challenge is to establish a solid foundation for software understanding, checking, and transformations. Static program analysis is an essential component of such a foundation. In this work we focus on a fundamental static analysis: *control-flow analysis*. Data-flow analysis is based on some control-flow representation, and our work also has direct implications for the design of data-flow analyses.

Traditional control-flow analysis cannot be directly applied to Android applications because they are framework-based and event-driven. The application code and the Android platform interact through *callbacks*: calls from the platform's event processing code to the relevant callback methods defined in the application code. A rich variety of callbacks is defined by the Android framework model, for events such as component creation/termination, user actions, device state changes, etc. We focus our analysis on a key aspect of this control flow: the lifecycle and interactions of *user-event-driven application components*. The first step of our work is to formulate this control-flow analysis problem in terms of the traditional concepts of interprocedural control-flow analysis [2], [3], thus providing a foundation for reasoning about run-time semantics and its static analysis approximations. In essence, the control-flow analysis problem can be reduced to modeling of the possible sequences of callbacks.

Next, we propose a program representation that captures such callback sequences. This representation, referred to as a *callback control-flow graph* (CCFG), together with the notion of *valid paths* in this graph, is the output of the control-flow analysis. We then present an algorithm for CCFG construction. The algorithm considers user-driven components such as activities, dialogs, and menus, and analyzes the corresponding lifecycle and event handling callback methods. The analysis of each callback method (and the code transitively invoked by it) determines what other callbacks may be triggered next. This information provides the basis for CCFG construction.

The key technical insight for the design of our algorithm is that a callback method must be analyzed *separately for different invocation contexts associated with it*. For example, an event handler method could process user events for several different widgets, and may have a different behavior for each separate widget. Our context-sensitive analysis employs a form of graph reachability that traverses context-compatible control-flow paths and identifies statements whose execution may trigger subsequent callbacks, as well as paths that avoid such statements. Through examples and experimental studies, we show the importance of this form of context sensitivity.

The proposed analysis can be a component of various other static analyses (e.g., for software checking [4]–[17]). We consider one such client: the automated generation of static GUI models, which are important for program understanding and test generation. The CCFG can be easily transformed into a certain kind of GUI model used in prior work (e.g., [18]–[21]), and possible sequences of user GUI events can be derived from valid paths in the model.

Using 20 applications, we performed an experimental evaluation whose results can be summarized as follows. First, the analysis cost is suitable for practical use in software tools. Second, the use of context-sensitive analysis results in substantial precision improvements. Third, in six case studies, we compared the produced GUI models against the "perfectly-precise" manually-constructed solution as well as the solution from a dynamic analysis tool [22]. This comparison indicates high analysis precision, with three of the case studies ex-

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.31

hibiting perfect precision, and the rest providing insights into potential precision improvements for future work.

Contributions. The contributions of this work are

- Formulation of the control-flow analysis problem for lifecycle and event handling callbacks
- Definition of the CCFG, a program representation that encodes the solution to this problem
- Algorithm to build the CCFG through context-sensitive control-flow analysis of callback methods
- Technique to build and traverse static GUI models based on the CCFG
- Evaluation on 20 applications, as well as detailed case studies using six of these applications

The CCFG definition and the analysis algorithms are novel contributions: to the best of our knowledge, this is the first effort to perform context-sensitive analysis of Android callback behavior and to encode the resulting control-flow information. This is also the first work to perform fully-static generation of GUI models for Android. The analysis implementation is publicly available as part of the GATOR analysis toolkit [23].

II. Control-flow Analysis for Android

Our work targets a fundamental problem: static control-flow analysis. Since data-flow analysis must model the program's control flow (in addition to the data-flow domain), control-flow analysis is also a key component of data-flow analysis.

A. Background

The standard program representation for such analysis is the *interprocedural control-flow graph* (ICFG). This graph combines the control-flow graphs (CFGs) of the program's procedures. Nodes correspond to statements, and intraprocedural edges show the control flow inside a procedure. The CFG for a procedure p has a dedicated start node s_p and a dedicated exit node e_p. Each call is represented by two nodes: a call-site node c_i and a return-site node r_i. There is an interprocedural edge $c_i \rightarrow s_p$ from a call-site node to the start node of the called procedure p; there is also a corresponding edge $e_p \rightarrow r_i$. An ICFG path that starts from the entry of the main procedure is *valid* if its interprocedural edges are matched (i.e., each r_i is matched with the corresponding c_i) [2], [3].

The goal of control-flow analysis is to determine the set of all valid paths. In an actual analysis, some abstractions of such paths are typically employed. Still, at its essence, control-flow analysis needs to find and abstract all valid paths.

For a framework-based platform such as Android, there is no main procedure from which control-flow paths start. The interaction between an application and the platform is through callbacks: the high-level view of the control flow is as a sequence of calls from (unknown) platform code to specific application methods. This is a key challenging aspect of Android control-flow analysis, and the focus of our work. Thus, we consider *abstracted* ICFG paths in which only interprocedural edges to/from callback methods are represented, and all other edges are abstracted away. In this case, a path consists of edges $c_i \rightarrow s_m$ and $e_m \rightarrow r_i$ where c_i is a call-site node in the platform code that invokes an application-defined callback method m, and r_i is the return-site node corresponding to c_i.

The Android framework defines thousands of callbacks for a variety of interactions. We focus on an essential aspect of this control flow: the lifecycle and interactions of user-event-driven components. The components of interest are *activities*, *dialogs*, and *menus*. Each such component is represented by a separate GUI window. We consider two categories of callbacks.

Lifecycle callbacks manage the lifetime of application components. The most important examples are callbacks to manage activities. Lifecycle methods such as `onCreate` and `onDestroy` are of significant interest because management of the activity lifecycle is an essential concern for developers (e.g., to avoid leaks [11], [12], [20]). Lifecycle callbacks for activities, dialogs, and menus define major changes to the visible state and to the possible run-time events and behavior.

GUI event handler callbacks respond to user actions (e.g., clicking a button) and define another key aspect of the control flow. These event handlers perform various actions, including transitions in the application logic (e.g., terminating an activity and returning back to the previous one). Control-flow analysis of such handlers is essential for an event-driven platform.

For these two categories of callbacks, the execution of a callback method m_i completes before any other callback method m_j is invoked. (As discussed later, m_i may cause the subsequent execution of m_j.) Thus, the abstracted control-flow paths are always of the form $c_i \rightarrow s_{m_i}, e_{m_i} \rightarrow r_i, c_j \rightarrow s_{m_j}, e_{m_j} \rightarrow r_j, c_k \rightarrow s_{m_k}, e_{m_k} \rightarrow r_k, \ldots$ and will be represented simply as $m_i\ m_j\ m_k \ldots$ where m_i is the callback method invoked by c_i. Thus, in this work we are interested in a version of control-flow analysis which produces all valid sequences of method callbacks for component lifecycles and event handling. We aim to model only a single application; inter-application control flow is beyond the scope of this work.

B. Example from an Android Application

Figure 1 shows a simplified example derived from OpenManager [24], an open-source file manager for Android. Class `Main` defines an activity: an application component responsible for displaying a GUI window and interacting with the user. Method `onCreate` is an example of a lifecycle callback method: it is invoked by the Android platform when the activity is instantiated. The structure of the new window is defined by file `main.xml` shown at the bottom of the figure. In this simplified example the layout contains four GUI widgets, each one being a button with an image that can be clicked. The call to `setContentView` at line 4 instantiates these widgets (together with their `LinearLayout` container) and associates them with the `Main` activity. The loop at lines 9–12 iterates over the programmatic button ids and associates the buttons with a listener object: the `EventHandler` created at line 5.

The listener class defines an *event handling method* `onClick`, which is invoked by the Android platform when the user clicks on a button. The button that was clicked is provided as parameter `v` of `onClick`. The event handler may start a new activity: an instance of `DirectoryInfo` (when `v`

```
1   public class Main extends Activity {
2      private EventHandler mHandler;
3      public void onCreate() {
4         this.setContentView(R.layout.main);
5         mHandler = new EventHandler(this);
6         int[] img_button_id = {R.id.info_button,
7            R.id.help_button, R.id.manage_button,
8            R.id.multiselect_button};
9         for(int i = 0; i < img_button_id.length; i++) {
10           ImageButton b = (ImageButton)findViewById(
11              img_button_id[i]);
12           b.setOnClickListener(mHandler);
13        } } }

14  public class EventHandler implements
15     OnClickListener {
16     private final Activity mActivity;
17     public EventHandler(Activity activity) {
18        mActivity = activity;
19     }
20     public void onClick(View v) {
21        switch(v.getId()) {
22           case R.id.info_button:
23              Intent info = new Intent(
24                 mActivity, DirectoryInfo.class);
25              mActivity.startActivity(info);
26              break;
27           case R.id.help_button:
28              Intent help = new Intent(
29                 mActivity, HelpManager.class);
30              mActivity.startActivity(help);
31              break;
32           case R.id.manage_button:
33              AlertDialog.Builder builder = ...
34              AlertDialog dialog = builder.create();
35              dialog.show();
36              break;
37           default:
38              ...
39              break; } } }

main.xml:
<LinearLayout>
  <ImageButton android:id="@+id/info_button"/>
  <ImageButton android:id="@+id/help_button"/>
  <ImageButton android:id="@+id/manage_button"/>
  <ImageButton android:id="@+id/multiselect_button"/>
</LinearLayout>
```

Fig. 1. Example derived from OpenManager [24]

is the info button, line 25) or of `HelpManager` (when `v` is the help button, line 30). In both cases, an `Intent` triggers the activation; this is the standard Android mechanism for starting a new activity. The call to `startActivity` posts an event on the framework's event queue. After `onClick` completes, this event is processed, a callback to `onCreate` is executed on the new activity, and a new window is displayed.

When `v` is the manage button, a new dialog window is created and displayed at line 35. This window is an instance of a `AlertDialog` and is used to show several selectable items (e.g., to manage the running process, or to back up applications to the SD card). The creation of the dialog is performed through helper object `builder`. Finally, when `v` is the multi-select button, the displayed window remains the one associated with activity `Main`, but its visual representation changes (line 38); details of this change are omitted.

Control-flow analysis for this application needs to capture the ordering relationship between `onCreate` and `onClick`: the event handler method may be invoked immediately after `onCreate` completes its execution. Similarly, control-flow analysis needs to capture the ordering relationship between `onClick` and `DirectoryInfo.onCreate`, `HelpManager.onCreate`, and `AlertDialog.onCreate`. In addition, because it is possible that the default branch of the switch statement is taken, the next callback after `onClick` could be another invocation of `onClick`.

Note that the flow of control triggered by `onClick` is *context sensitive*: depending on the widget (parameter `v`), different sequences of callbacks may be observed. From prior work on control-flow/reference analysis of object-oriented programs (e.g., [25]–[27]), it is well known that context sensitivity has significant precision benefits. One effective way to introduce context sensitivity is to model the parameters of a method invocation (including `this`) [25]. Based on this observation, we propose a new form of context-sensitive control-flow analysis of callback methods. For this example, a context-insensitive analysis would conclude that the execution of `onClick` could be followed by execution of any one of the other four callbacks. However, a context-sensitive analysis will report that, for example, `onClick` will be followed by `HelpManager.onCreate` only when `v` was the help button.

C. Problem Definition

Consider two sets of application methods: set $\mathcal{L}$ of lifecycle methods for activities, dialogs, and menus, as well as set $\mathcal{H}$ of GUI event handler methods. Sequences of callbacks to such methods are the target of our analysis. In this work we focus on certain lifecycle methods $l \in \mathcal{L}$: specifically, *creation* callbacks (e.g., `Activity.onCreate`) and *termination* callbacks (e.g., `Activity.onDestroy`).

In the future we plan to extend the analysis with other lifecycle callbacks based on standard ordering constraints [11]. In addition, callbacks related to other types of components—such as services, broadcast receivers, and non-UI worker threads—are of great interest to the static analysis community (e.g., for security checking and leak detection), and context-sensitive analysis for such callbacks is an important target for future work. Such generalizations would require significant conceptual extensions to our current analyses.

We assume that relevant static abstractions have already been defined by an existing analysis of GUI-related objects [28], [29]. We will refer to this analysis as GATOR, using the name of its public implementation [23]. The analysis tracks the propagation of widgets and related entities (e.g., activities, dialogs, listeners, layout/widget ids) by analyzing XML layouts and relevant code (e.g., the calls to `findViewById` and `setOnClickListener` in Figure 1). Its output contains a pair of sets $(\mathcal{W}, \mathcal{V})$. Each window $w \in \mathcal{W}$ (an activity, a dialog, or a menu) is associated with a set of views $v \in \mathcal{V}$. Views are the Android representation of GUI widgets, and are instances of subclasses of `android.view.View`. A widget $v \in \mathcal{V}$ may be associated with event handlers $h \in \mathcal{H}$.

The control-flow analysis output can be represented by a *callback control-flow graph* (CCFG). There are three categories of graph nodes. A node $(h, v) \in \mathcal{H} \times \mathcal{V}$ indicates that event handler h was executed due to a GUI event on widget v. A node $(l, w) \in \mathcal{L} \times \mathcal{W}$ shows that lifecycle method l was

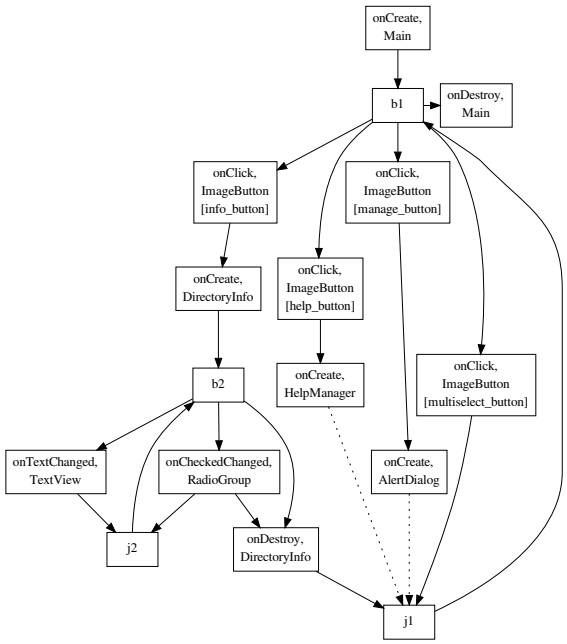

Fig. 2. Callback control-flow graph

executed on window w. In addition, helper nodes are used to represent branch and join points, as explained shortly. The start node in the CCFG corresponds to the `onCreate` callback on the main activity of the application. Each path starting from this node defines a possible sequence of callbacks during the execution of the application. An edge $n_1 \rightarrow n_2$ shows that the callback represented by n_1 may trigger the subsequent execution of the callback represented by n_2.

The CCFG for the running example is shown in Figure 2. For illustration, we show a scenario where (1) the main activity also has an `onDestroy` lifecycle method, (2) the details of `HelpManager` and `AlertDialog` are not elaborated, and are represented by the two dashed edges, (3) `DirectoryInfo` has two event handlers as well as an `onDestroy` method, and (4) handler `onCheckedChanged` may force termination of `DirectoryInfo` and return control back to `Main`.

To indicate that event handlers could be executed in any order, branch nodes b_i and join nodes j_i are introduced, together with edges $j_i \rightarrow b_i$. This technique is similar to our early work on data-flow analysis approximations [30]; recent work [9] also uses a similar approach, as discussed later. Both `onDestroy` methods are successors of the corresponding branch nodes (rather than join nodes) to show that the user may click the device's BACK button to exit an activity immediately, without triggering any event handler. Note that `onDestroy` in `DirectoryInfo` is also a successor of `onCheckedChanged`, to show that this handler may force exit from `DirectoryInfo` (e.g., by using a standard API call such as `finish`).

This model is not complete: for example, if `Main` is the current window and the screen is rotated, a new instance of `Main` will replace the current one, and `onCreate` would be called on it, which would require additional edges in the graph. Such edges can be added for standard Android events such as screen rotation, interruption due to a phone call, or locking/unlocking the device screen [29]; for simplicity, we do not consider them in this paper.

D. Prior Work

Existing work has addressed some aspects of this problem. For example, FlowDroid [9] uses a static analysis that represents the possible orderings of lifecycle/event callbacks for a single activity. The analysis encodes these orderings in an artificial main method, and paths through this method correspond to sequences of callbacks. This approach was designed for a particular form of interprocedural taint analysis and does not solve the general control-flow problem described above. The key issue is that there is no modeling of transitions and interactions involving *multiple activities*. For example, there is no path through the main method to show that the execution of `EventHandler.onClick` may trigger the execution of `DirectoryInfo.onCreate`; the same is true for the other two `onCreate` methods. In addition, the approach does not consider the widgets on which the event handlers operate, nor does it model transitions to/from dialogs and menus, or transitions due to window termination. The earlier SCanDroid tool [4], which aims to model the sequence of callbacks to event handlers [31], has similar limitations.

Another area of related work is the resolution of activity-launch calls, such as the `startActivity` calls at lines 25 and 30 in Figure 1. Activity-launch APIs use an intent object to specify the target activity; two examples are shown at lines 23 and 28 in the figure. There are several existing techniques [4], [5], [8], [14], [15], [32] for analysis of intent objects. By itself, intent analysis cannot determine the edges in a CCFG (shown in Figure 2). It needs to be combined with (1) context-sensitive analysis of event handlers and their transitive callees, (2) tracking of other window-launch calls (e.g., the call to `show` at line 35), and (3) modeling of window termination calls. One component of our control-flow analysis is an intent analysis which is derived from prior work [8].

III. Analysis Algorithm

A. Control-Flow Analysis of a Callback Method

A key building block of our approach is a context-sensitive analysis of a callback $m \in \mathcal{L} \cup \mathcal{H}$ under a context c. Recall that we use static abstractions for windows $w \in \mathcal{W}$ (activities, dialogs, and menus) and views $v \in \mathcal{V}$ created by GATOR. For an event handler $h \in \mathcal{H}$, the context is a view v; for a lifecycle callback $l \in \mathcal{L}$, the context is a window w. The analysis is outlined in Algorithm 1. This algorithm is then used by the main control-flow analysis, as described in Section III-B.

Input and output. The algorithm traverses valid ICFG paths, starting from the entry node of m's CFG. When a *trigger node* is reached, the traversal stops. A trigger node is a CFG node that may trigger the subsequent execution of another callback; the set $triggerNodes$ of all such nodes is provided as input to the algorithm. Examples of trigger nodes are shown at lines

Algorithm 1: AnalyzeCallbackMethod(m,c)

```
   Input: m : callback method
   Input: c : context
   Input: triggerNodes : set of ICFG nodes
   Output: reachedTriggers ← ∅ : set of ICFG nodes
   Output: avoidsTriggers : boolean
1  feasibleEdges ← ComputeFeasibleEdges(m, c)
2  visitedNodes ← {entryNode(m)}
3  nodeWorklist ← {entryNode(m)}
4  avoidingMethods ← ∅
5  while nodeWorklist ≠ ∅ do
6      n ← removeElement(nodeWorklist)
7      if n ∈ triggerNodes then
8          reachedTriggers ← reachedTriggers ∪ {n}
9      else if n is not a call-site node and not an exit node then
10         foreach ICFG edge n → k ∈ feasibleEdges do
11             Propagate(k)
12     else if n is a call-site node and
       n → entryNode(p) ∈ feasibleEdges then
13         Propagate(entryNode(p))
14         if p ∈ avoidingMethods then
15             Propagate(returnSite(n))
16     else if n is exitNode(p) and p ∉ avoidingMethods then
17         avoidingMethods ← avoidingMethods ∪ {p}
18         foreach c → entryNode(p) ∈ feasibleEdges do
19             if c ∈ visitedNodes then
20                 Propagate(returnSite(c))
21 avoidsTriggers ← m ∈ avoidingMethods
22 procedure Propagate (k)
23     if k ∉ visitedNodes then
24         visitedNodes ← visitedNodes ∪ {k}
25         nodeWorklist ← nodeWorklist ∪ {k}
```

25, 30, and 35 in Figure 1; other examples are provided in Section III-B. An analysis output is the set $reachedTriggers$ of trigger nodes encountered during the traversal.

Another key consideration is to determine whether the exit node of m is reachable from the entry node of m via a valid trigger-free ICFG path. If so, the execution of m may avoid executing any trigger. In the example such a path exists through the default branch. This path is necessary to determine the CCFG edge from `onClick` to j_1 for the multiselect button. This edge shows that when this button is clicked, `onClick` may be followed by another invocation of `onClick` (or by app termination). The algorithm outputs a boolean $avoidsTriggers$ indicating the existence of a trigger-free path.

Context sensitivity. Context sensitivity is achieved by performing a separate pre-analysis—represented by the call to ComputeFeasibleEdges—to determine the feasible ICFG edges in m and methods transitively called by m. During the traversal (lines 5–20 in Algorithm 1), only feasible edges are followed. The choice of the feasibility pre-analysis depends on the callback method and on the context. For example, when `onClick` from the running example is analyzed, the context is a static abstraction of the `ImageButton` instance provided as parameter. Using the output from Gator, the id of this view is also available. This allows ComputeFeasibleEdges to resolve the return value of `v.getId()` at line 21 and to determine which branch is feasible. The general form of this pre-analysis is outlined in Section III-C. For a lifecycle callback under the context of a window, the analysis can identify virtual calls where this window is the only possible receiver, and can determine more precisely the feasible interprocedural edges.

Algorithm design. The algorithm is based on the general graph-traversal technique for solving interprocedural, finite, distributive, subset (IFDS) data-flow analysis problems [3]. We formulated an IFDS problem with a lattice containing two elements: $\emptyset$ and the singleton set $\{entryNode(m)\}$. The data-flow functions are $\lambda x.x$ (identity function, for non-trigger nodes) and $\lambda x.\emptyset$ (for trigger nodes). The resulting data-flow analysis was the conceptual basis for Algorithm 1.

Set $avoidingMethods$ contains methods p that are proven to contain a trigger-free same-level valid path from the entry of p to the exit of p. (In a same-level valid path, a call site has a matching return site, and vice versa.) Thus, the execution of p may avoid any trigger. If a call-site node is reachable, and it invokes such a method, the corresponding return-site node is inferred to be reachable as well (lines 14–15). As another example, whenever the exit node of p is reached for the first time (line 16), p is added to $avoidingMethods$ and all call sites c that invoke p are considered for possible reachability of their return sites (lines 18–20). The set of avoiding methods is, in essence, a representation of the IFDS summary edges [3].

B. CCFG Construction

CCFG construction uses the output from Gator. In this output, an activity a is associated with widgets $Views(a) \subseteq \mathcal{V}$. The activity could also be associated with an options menu $m \in \mathcal{W}$; such a menu is triggered by the device's dedicated menu button or by the action bar. Similarly, a view $v \in \mathcal{V}$ could have a context menu m, triggered through a long-click on the view. Each menu m represents a separate window with its own widget set $Views(m)$, which typically contains views (instances of `MenuItem`) representing items in a list. A dialog $d \in \mathcal{W}$ is a separate window with some message to the user, together with related choices (e.g., buttons for "OK" and "Cancel"). A dialog is associated with its own widget set $Views(d)$. A widget v could be associated with several event handlers $Handlers(v) \subseteq \mathcal{H}$.

CCFG construction creates, for each $w \in \mathcal{W}$, nodes for the relevant callbacks. Lifecycle methods for creation and termination of w are based on standard APIs. In the subsequent description we assume that w defines both a creation callback l^c (e.g., `onCreate`) and a termination callback l^t (e.g., `onDestroy`), but our implementation does not make this assumption. For any $h \in Handlers(v)$ where $v \in Views(w)$, there is a CCFG node (h, v); we assume that at least one such node exists for w. A branch node b_w and a join node j_w are also introduced.

Edge creation. Algorithm 2 defines the edges created for a window w. As illustrated in Figure 2, edges $(l^c, w) \rightarrow b_w \rightarrow (l^t, w)$ show the invocations of lifetime callbacks; these edges are created at lines 5–6 in Algorithm 2. The second edge represents the termination of w with the BACK button.

Algorithm 2: CreateEdges(w)

Input: w : window
Input: $(l^c, w), (l^t, w)$: lifecycle nodes for w
Input: $\{(h_1, v_1), (h_2, v_2), \ldots\}$: event handler nodes for w
Input: b_w, j_w : branch/join nodes for w
Output: $newEdges$: set of CCFG edges for w

```
1  newEdges ← ∅
2  ⟨triggers, avoids⟩ ← AnalyzeCallbackMethod(l^c, w)
3  newEdges ← newEdges ∪ TriggerEdges(triggers, l^c, w)
4  if avoids then
5      newEdges ← newEdges ∪ {(l^c, w) → b_w}
6  newEdges ← newEdges ∪ {b_w → (l^t, w)}
7  foreach event handler node (h, v) do
8      newEdges ← newEdges ∪ {b_w → (h, v)}
9      ⟨triggers, avoids⟩ ← AnalyzeCallbackMethod(h, v)
10     newEdges ← newEdges ∪ TriggerEdges(triggers, h, v)
11     if avoids then
12         newEdges ← newEdges ∪ {(h, v) → j_w}
13 if w is not a menu then
14     newEdges ← newEdges ∪ {j_w → b_w}
15 else
16     newEdges ← newEdges ∪ {j_w → (l^t, w)}
```

The termination of w could also be triggered by event handlers. Recall that for the running example, we assume that handler `onCheckedChanged` calls `finish` on activity `DirectoryInfo`. This is an example of a termination trigger node, and our analysis creates an edge from `onCheckedChanged` to `onDestroy` (at line 10 in Algorithm 2, as elaborated below). Furthermore, if the handler's execution cannot avoid this trigger, the analysis would *not* create the edge from `onCheckedChanged` to j_2. For the example, we assume that this termination trigger can be avoided along some ICFG path; thus, there is an edge to j_2 in Figure 2, created by line 12 in Algorithm 2.

For each handler h for a view v from w's widget set, an edge $b_w \rightarrow (h, v)$ is added to indicate the possible user actions and the invoked handlers (line 8 in Algorithm 2). Together with the back edge $j_w \rightarrow b_w$ created at line 14, this structure indicates arbitrary ordering of user-triggered events. If w is a menu, menu item selection immediately closes w and an edge from j_w to the termination callback is created instead.

Each h is analyzed under context v using Algorithm 1 (invoked at line 9). If $avoidsTriggers$ is true, $(h, v) \rightarrow j_w$ is added to show that the execution of h may retain the current window w (rather than transition to a new one) and user events will continue to trigger the event handlers for w. The other outgoing edges for (h, v) are determined by set $reachedTriggers$ and are created by helper function TRIGGEREDGES described below.

Algorithm 1 is also invoked for the creation callback l^c (at line 2) to determine which trigger statements are reachable. Termination callback l^t is assumed to contain no such triggers, since its role is to clean up resources, rather than to trigger window transitions. Edge creation for (l^c, w), shown at lines 3–5, is similar to the edge creation for (h, v) at lines 10–12.

The set of edges produced by TRIGGEREDGES is based on case-by-case analysis of trigger statements. Activity-launch calls such as `startActivity` (e.g., lines 25 and 30 in Figure 1) are analyzed with our flow- and context-insensitive intent analysis, conceptually derived from a more expensive prior analysis [8], but accounting for statement feasibility (analogous to line 1 in Algorithm 1). The analysis focuses on explicit intents because they are designed for use inside the same application [33]. In our experience, it performed as well as existing alternatives [8], [31]. Menu-launch calls such as `showContextMenu` as well as dialog-launch calls (e.g., line 35 in Figure 1), are resolved by GATOR. Any such statement triggers the launch of a new window w'. Correspondingly, function TRIGGEREDGES produces edges $(h, v) \rightarrow (l'^c, w')$ and $(l'^t, w') \rightarrow j_w$ when invoked at line 10, and edges $(l^c, w) \rightarrow (l'^c, w')$ and $(l'^t, w') \rightarrow b_w$ when invoked at line 3.

TRIGGEREDGES also accounts for the possibility that set $triggers$ contains a statement that terminates the current window—e.g., a call to `finish` on an activity, or a call to `dismiss` on a dialog. If $triggers$ contains a termination statement for w, TRIGGEREDGES produces $(h, v) \rightarrow (l^t, w)$ or $(l^c, w) \rightarrow (l^t, w)$ to represent the possible flow of control.

Example. For the running example shown in Figure 1, calling ANALYZECALLBACKMETHOD at line 9 with $h =$ `onClick` and $v =$ `ImageButton[info_button]` will return $triggers = \{s_{25}\}$ and $avoids = false$. Activity-launch statement s_{25}, representing line 25 in Figure 1, is resolved to $w' =$ `DirectoryInfo`. As a result, edges $(h, v) \rightarrow$ (`onCreate`, w') and (`onDestroy`, w') $\rightarrow j_1$ are produced by TRIGGEREDGES. If the call at line 9 is for $h =$ `onCheckedChanged`, $triggers$ will contain the call to `finish` that closes w', resulting in an edge to (`onDestroy`, w') created by TRIGGEREDGES.

C. Detection of Feasible Edges

To detect which ICFG edges are (in)feasible under a particular context, we use constant propagation analyses formulated as Interprocedural Distributed Environment (IDE) [34] analysis problems. Due to space limitations, we outline the analyses at a high level without providing all technical details.

Consider the analysis of a callback method m under context c, performed by COMPUTEFEASIBLEEDGES. The context could be a widget v or a window w; both cases are handled, although our experiments suggest that context sensitivity for windows w has minor effect on precision. First, this analysis uses a form of interprocedural constant propagation to identify each local variable that definitely refers to only one object. This analysis employs (1) knowledge that a particular parameter of m definitely refers to c, and (2) additional reference information obtained from GATOR. The analysis considers all methods transitively invoked by m; virtual calls are resolved using class hierarchy information. After this constant propagation, the computed information is used to refine virtual call resolution: if only one receiver object is determined to be possible, the call is resolved accordingly. Next, another interprocedural constant propagation analysis determines constant values of integer and boolean type. For example, for an API call such as `x.getId()` or `x.getMenuItemId()`, if `x` definitely refers only to one particular view, the id for that view is treated as the

return (constant) value of the call. Boolean expressions such as `x==y` and `x!=y` are also considered, both for references and for integers; switch statements are treated similarly. In a final step, branch nodes whose conditions are found to be constants are used to determine infeasible ICFG edges, which (together with infeasible interprocedural edges at refined virtual calls) defines the output of COMPUTEFEASIBLEEDGES.

Example. For the example in Figure 1, suppose we analyze `onClick` under context `ImageButton[info_button]`. The first constant propagation analysis will determine that `v` definitely points to only this button. The second constant propagation analysis will determine that `v.getId()` returns the integer constant `R.id.info_button`. The output of the analysis will be the set of ICFG edges corresponding to the first branch of the switch statement. Although in this simple example the propagation is trivial, our analyses handle general interprocedural propagation along valid ICFG paths, using jump functions and summary functions [34].

D. Valid CCFG Paths

Not every path in the CCFG represents a valid sequence of run-time invocations of callback methods. Consider again the example in Figure 2, and suppose that another window w, different from the ones shown in the figure, contained a handler h with `startActivity` call to trigger window $w' =$ `DirectoryInfo`. Edges $(h, v) \rightarrow (\texttt{onCreate}, w')$ and $(\texttt{onDestroy}, w') \rightarrow j_w$ would be created to represent this trigger statement. Clearly, a path that enters w' from `Main` through edge $(\texttt{onClick}, \texttt{info\_button}) \rightarrow (\texttt{onCreate}, w')$, but exits w' back to w, through $(\texttt{onDestroy}, w') \rightarrow j_w$, does not correspond to a valid run-time execution.

In general, a *valid* CCFG path has matching edges $\ldots \rightarrow (l^c, w)$ and $(l^t, w) \rightarrow \ldots$. Each such pair is created at the same time by TRIGGEREDGES and can be recorded as a matching pair at that time. This condition is very similar to the traditional one for valid ICFG paths. The implications for static analyses are also similar to traditional ICFG control-flow analysis. Standard techniques can be applied to focus only on valid CCFG paths (or some over-approximation): either by explicitly maintaining the sequence of unmatched $\ldots \rightarrow (l^c, w)$ edges, or by creating approximations of them, in the spirit of k-call-site-string sensitivity [2], [25].

IV. CLIENT ANALYSIS: CONSTRUCTION OF GUI MODELS

The control-flow analysis described above could be used as a component of other static analyses (e.g., [4]–[17]). These potential clients provide a number of interesting possibilities for future work. Another possible use of the analysis is for *generation of GUI models*, which are important for program understanding and test generation. Various GUI models have been employed (e.g., [18], [22], [35], [36]). Figure 3 shows an example of the GUI models we consider. Nodes represent GUI windows, and edges show possible transitions between windows. Each transition is labeled by the GUI widget that triggers it. Additional information for an edge is the type of GUI event (e.g., click) and the event handler (e.g., method `onClick`). For simplicity, this information is not displayed in Figure 3. "BACK" edges correspond to the device's back button. "Launcher" denotes the Android app launcher. Only a subset of the edges are shown: for example, the edge from `HelpManager` back to `Main` is not shown.

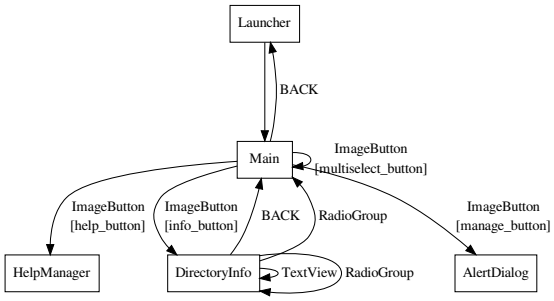

Fig. 3. GUI model for the running example

Such a model can serve as starting point for test generation. For example, an existing automated test generation technique for Android [19] requires the set of tuples (window w, GUI widget v, event e, handler method h), where v is visible when w is active, and event e on v is handled by h. In this earlier work such models are constructed manually. As another example, a test generation approach for exposing leaks in Android applications [20] requires this model as input; in that work the models were also manually created. There are other examples of model-based test generation for Android where a GUI model is an essential prerequisite [18], [21], and paths in the model correspond to GUI events in a test case.

This GUI model can be easily derived from the CCFG. Edges to a creation callback (l^c, w) in the CCFG represent transitions to window w in the model; the sources of these edges describe the event handlers and widgets. Edges from a termination callback (l^t, w) represent returns from w to the previous window. In some cases, the return is not to the previous window, but to an earlier one; our analysis considers which previous windows may be already closed, and direct the edges accordingly. The predecessors of (l^t, w) describe which events trigger the return; when the predecessor is b_w, the event is "BACK". Self-transitions in the GUI model (e.g., for `multiselect_button` in Figure 3) are also easy to derive.

Each path in this model corresponds to a unique CCFG path. A path in the model is valid only if its corresponding CCFG path is valid. If the model is traversed to create test cases (e.g., as in [20]), only valid paths should be traversed.

Our context-sensitive analysis is needed to avoid infeasible edges in the GUI model. If a context-*in*sensitive analysis were used instead, it would conclude that the execution of `onClick` could trigger each one of the four possible edges in the model, regardless of the button being clicked. As a result, for example, infeasible edges for `manage_button` to `HelpManager` and `DirectoryInfo` would be created, as well as an infeasible self-edge to `Main`. Overall, twelve infeasible edges would be added to the GUI model for the running example.

TABLE I
CHARACTERISTICS OF THE ANALYZED APPLICATIONS AND THEIR CCFGS.

(a) Applications						(b) CCFGs				(c) Models		(d) Times	
Name	Classes	Methods	Activities	Menus	Dialogs	Nodes	Edges	OutDegree	OutDegCI	OutDegree	OutDegCI	CS [s]	CI [s]
APV	68	413	4	4	5	88	158	1.15	2.98	4.23	10.85	10	8
Astrid	1228	5782	41	3	48	980	1896	1.14	1.14	12.32	12.36	105	54
BarcodeScanner	126	594	9	4	6	104	171	1.37	1.88	3.28	4.44	10	10
Beem	284	1883	12	6	5	121	186	1.14	2.20	2.57	5.24	12	12
ConnectBot	371	2366	11	8	17	197	317	1.20	1.20	3.20	3.20	28	22
FBReader	954	5452	27	9	8	272	2916	11.41	12.59	55.95	61.67	843	269
K9	815	5311	32	3	19	393	723	1.15	1.59	5.90	7.96	79	54
KeePassDroid	465	2784	20	11	9	288	682	2.01	2.47	12.82	17.18	27	20
Mileage	221	1223	50	15	9	522	914	1.34	1.70	5.41	6.64	13	12
MyTracks	485	2680	32	8	20	286	630	1.83	4.31	8.53	18.71	23	21
NPR	249	1359	13	12	6	564	1175	1.19	2.08	34.29	59.29	16	15
NotePad	89	394	8	3	10	134	259	1.32	2.84	5.48	10.38	9	9
OpenManager	53	237	6	2	9	110	183	1.10	2.30	4.31	9.06	7	6
OpenSudoku	140	726	10	6	18	170	307	1.41	3.44	3.12	7.26	11	11
SipDroid	331	2863	12	5	13	148	346	2.00	3.98	8.47	15.10	39	34
SuperGenPass	64	267	2	2	4	61	107	1.18	1.64	5.00	6.88	8	8
TippyTipper	57	241	6	3	0	61	94	1.00	1.24	4.13	5.13	6	6
VLC	242	1374	10	2	13	169	278	1.10	1.10	4.24	4.24	19	18
VuDroid	69	385	3	2	1	35	62	1.50	3.33	3.67	8.00	5	5
XBMC	568	3012	22	20	24	2275	6254	1.85	2.24	176.07	186.33	38	32

V. EXPERIMENTAL EVALUATION

We applied the analysis on 20 applications used in prior work [17], [20], [28], [36], [37]. Our goals were to (1) characterize the size and complexity of the CCFG, (2) measure the benefits of context sensitivity, and (3) evaluate the precision of the GUI models derived from the CCFG.

A. CCFG Construction

Table I(a) shows the number of application classes and methods, as well as counts for different categories of windows. Typically, an analyzed application has more than twenty windows. The callback sequences and GUI models associated with these windows cannot be practically analyzed by hand. This complexity is also indicated in part (b) of the table: there are typically more than a hundred CCFG nodes for an application, where each node (except for branch/join nodes) represents a callback under a particular context.

Column "OutDegree" shows the average number of outgoing edges for CCFG nodes corresponding to event handlers. This number is an indication of the variability of behavior for a handler (e.g., `onClick` in the running example). Our context-sensitive analysis aims to model this variability more precisely, by accounting for the handler's context. To measure the effects of context sensitivity, we also ran the analysis in a context-insensitive mode, where context information was ignored (i.e., the call to COMPUTEFEASIBLEEDGES in Figure 1 was not used). Column "OutDegCI" shows the resulting average number of outgoing edges. As the measurements in the two columns show, there can be significant precision loss if context sensitivity is not used. This observation is confirmed by part (c), which contains similar measurements for the average number of outgoing edges for a node in the static GUI model. (Back button edges were not included in these measurements, since each forward edge implicitly has a corresponding back button edge.) More generally, these results indicate that callback analysis with context sensitivity produces a more precise representation of the control flow. Section V-B provides additional case studies on the benefits of context sensitivity for GUI models.

We investigated two programs with significantly higher outdegrees measurements in part (c), compared to the rest of the programs: `FBReader` and `XBMC`. For `FBReader`, the culprit is a utility method that is called by the handlers of about 37% of the CCFG nodes. These nodes have significantly higher outdegrees than the rest. We examined a sample of such nodes, and determined that around 60% of their outgoing edges are feasible. The infeasible edges are due to the utility method: in it, class hierarchy analysis is used to resolve a `run()` call on a thread, which is overly conservative. While not comprehensive, this examination indicated that `FBReader` has a rich GUI with complex logic in event handlers, and this leads to a large number of possible window transitions. For `XBMC`, the large number of edges is due to a known imprecision of GATOR for this program [28]: because the analysis is context-insensitive, there is spurious propagation of widgets.

Part (d) of the table shows the running times of the CCFG construction analysis (including GATOR analyses), both for context-sensitive (CS) and context-insensitive (CI) algorithms. The running times of the GUI model construction are not shown, since they were negligible. Overall, the results indicate that analysis running times are suitable for practical use in software tools. The use of context-insensitive analysis typically does not lead to significant reductions in running time, and the resulting precision loss does not seem justified.

B. Case Studies of GUI Model Construction

To obtain additional insights on the precision of the static GUI models, case studies were performed on six applications:

TABLE II
EDGES IN THE GUI MODEL.

Application	Static (CS/CI)	Precise	Ripper	Ripping time
APV	55/141	55	22	1h34m
BarcodeScanner	59/80	43	20	4h44m
OpenManager	69/145	56	43	6h51m
SuperGenPass	40/55	40	19	1h26m
TippyTipper	33/41	33	28	1h21m
VuDroid	22/48	18	14	44m

APV, BarcodeScanner, OpenManager, SuperGenPass, TippyTipper, and VuDroid. These applications have the smallest number of windows in Table I, and were chosen to allow comprehensive manual examination.

We compared our static approach with Android GUI Ripper [22] ("Ripper" for short), a state-of-the-art tool for automated dynamic exploration of an application's GUI.[1] The public version from the tool's web page [39] was used for these experiments. The ripping observes run-time widgets for the current window, and fires events on them to cause GUI changes. If a new GUI state is discovered, its widgets are also considered for further events. In our experiments we let Ripper run to completion; the running times are shown in the last column of Table II. Each transition triggered during the dynamic exploration was mapped to a window-to-window transition edge, similar to the ones in the static model.

The results of this experiment are summarized in Table II. We first determined the set of GUI model edges based on CCFG construction (column "Static"). For precision comparison, we present results for both context-sensitive (CS) and context-insensitive (CI) construction of the CCFG. Next, we performed a careful case study of each application. Each GUI model edge reported by the context-sensitive analysis was manually classified as "feasible" or "infeasible". The number of feasible edges is shown in column "Precise".

To determine this number, we tried to manually achieve run-time coverage of the edge by triggering a transition from the source window to the target window, using the widget and GUI event for this edge. For infeasible edges, the source code was examined to determine that no run-time execution could cover this edge. Clearly, this manual examination presents a threat to validity; to reduce this threat, the code was examined by multiple co-authors. Below we describe details of some of these studies, as they shed light on the sources of imprecision of the proposed analysis. Column "Ripper" shows how many of the feasible edges could have been inferred from the dynamic exploration performed by Ripper.

The following conclusions can be drawn from these results. First, the overall precision of the context-sensitive static analysis is quite good. This leads to a small number of infeasible edges, which is beneficial for program understanding tools and testing tools (e.g., to compute more precise GUI coverage metrics). Second, one of the reasons for the good precision is the use of context sensitivity. A context-insensitive analysis would have increased the number of infeasible edges by more than a factor of 8. Third, the dynamic exploration in Ripper can miss significant portions of the GUI. One reason is that the exploration order may affect which widgets are available for interaction. For example, in APV, buttons "Clear Find", "Find Prev", and "Find Next" will not be available until a search action is finished, and "Find Prev" and "Find Next" are not available after "'Clear Find" is clicked. As another example, in OpenManager, if a file is deleted before being copied, related widgets and edges will be missed. As usual, static and dynamic approaches both have their respective strengths and weaknesses. For example, run-time state can be used to create a finer-grain dynamic GUI model, with multiple nodes for the same activity (based on widget states), which could potentially improve program understanding and test generation.

For the three applications with infeasible edges, we performed manual analysis to understand the sources of imprecision. Some examples of such sources are described below.

VuDroid. This application displays PDFs and DjVu files. Class BaseViewerActivity defines several event handlers shared by its two subclasses PdfViewerActivity and DjvuViewerActivity. One of these handlers restarts the current activity in full-screen mode by reusing the activity's intent. Since the handler is in the superclass, in our intent analysis both intents flow to the "restart" call, and it appears that each activity can trigger the other one, which cannot actually happen at run time. This explains all four infeasible edges in the model. If the event handlers in the superclass were cloned in the subclasses, the imprecision would be eliminated.

BarcodeScanner. This application scans and processes eleven types of barcodes. Depending on the barcode type, various GUI widgets are displayed to the user. For example, for one particular group of buttons, one subset of the group is used for an address book barcode, while a different subset is used for an email barcode. GATOR cannot distinguish statically which subsets are enabled for different barcode types, and concludes that all buttons in the group are always used, and that all eleven handlers may be invoked for each button. This imprecision is responsible for 13 out of the 16 infeasible edges. It seems unlikely that GATOR can be generalized to handle this case, since it would require an intricate combination of reference analysis with context-sensitive treatment of formals, together with interprocedural constant propagation for integers, and loop unrolling for constant-bound loops. An intriguing possibility for such cases is a hybrid static/dynamic approach.

OpenManager. For this application, the main source of imprecision is the context-insensitive nature of GATOR. The main activity of the application creates a dialog object and initializes it in a switch statement. Different branches of the switch correspond to different dialog layouts and widgets. In the analysis, all these widgets are associated with that one dialog object. The switch is based on the value of an integer formal parameter, which defines the calling context of the surrounding method. However, GATOR does not employ

[1]We initially also considered another dynamic exploration tool [32] but observed that sometimes it achieved lower GUI coverage. We also attempted to obtain the reverse engineering tool used in [36], but its proprietary implementation could not be distributed outside of the company [38].

this context information. If the application code were slightly different, with a separate dialog object being created for each context, the number of edges in the model would be reduced from 69 to 57. It would be interesting to consider context-sensitive generalizations of GATOR, employing ICFG traversal techniques similar to the ones used for CCFG construction.

VI. Related Work

Control-flow analysis for Android. Static analysis to understand GUI-driven behavior is essential for modeling the control/data flow of Android applications. The early work on the SCanDroid security analysis tool [4], [31] includes control-flow analysis and security permissions analysis for activities and other Android components (e.g., background services). The tool performs intent analysis and determines the inter-component control flow based on it. The approach does not employ static modeling/analysis for GUI objects, events, and handlers that trigger the inter-component transitions, and uses conservative assumptions about the GUI-related control/data flow. Subsequent work on related security problems, which also uses intent analysis and control-flow analysis [5], [6], [8], has similar limitations. As described in Section II-D, and indicated by our experiments, comprehensive and precise control-flow analysis requires context-sensitive analysis of event handlers and the actions taken by them (e.g., component creation and termination).

SCanDroid's static analysis is used in the A^{3}E tool [32] to construct an activity transition graph, which subsequently guides run-time GUI exploration. In this graph nodes correspond to activities and edges indicate transitions between them. It is unclear how this analysis models arbitrary GUI objects and handlers associated with an activity, and how these handlers are analyzed. Similar limitations exist for a static/dynamic analysis of UI-based triggers [7], where security-sensitive behaviors are triggered dynamically based on a static model of activity transitions. The static model construction in this work is incomplete, since it is based on restrictive assumptions about event handlers and on rudimentary analysis of these handlers. Our control-flow analysis provides a more rigorous and general solution to this problem.

FlowDroid [9] is a precise flow- and context-sensitive taint analysis which performs interprocedural control-flow and data-flow analysis for Android. As part of this approach, the effects of callbacks are modeled by creating a wrapper main method. Our CCFG is conceptually similar, but without explicitly creating a wrapper. In FlowDroid, artificial placeholder GUI objects that may flow into these callbacks are created in the wrapper method, while our approach propagates to the callbacks the actual GUI objects. The tool does not model the general propagation of GUI widgets and listeners, nor does it analyze event handlers context sensitively. As discussed in Section II-D, this analysis does not represent control flow that spans multiple activities. In particular, information of the form "callback m_i may be followed by callback m_j" cannot be inferred when m_i and m_j do not belong to the same activity. Control flow involving dialogs, menus, and window termination are also not handled. Providing the CCFG as input to FlowDroid is an intriguing possibility for future work.

In CHEX [13], each callback method and all its transitive callees are defined as a code split, and split permutations are used to derive the set of control-flow paths. Another security analysis [40] also considers all possible permutations of callbacks. AsDroid [14] analyzes event handlers of GUI objects to detect stealthy behaviors, but does not systematically model the GUI objects and their handlers, nor does it account for the widget context of a handler. A proposed operational semantics for activities [41] captures aspects of Android control flow, but does not model associations between widgets and handlers, and does not develop analysis algorithms. Apposcopy [15] builds an inter-component call graph, as part of a malware detection analysis, but it is unclear how it models GUI widgets, events, and handlers. These existing techniques could potentially be complemented by our CCFG.

The CCFG representation may be able to serve as basis for data-flow analyses to check statically various other properties of Android applications—for example, the absence of leaks [11], [12], [20]. Other examples include energy-related defects (e.g., [10] needs to model the execution order of handlers), errors due to invalid thread accesses to GUI objects [16], and errors related to null references and non-termination [17].

GUI models for Android. Reverse engineering of GUI models has been studied by others (e.g., [42]–[44]) and has been applied to Android (e.g., [22], [32], [36], [39], [45]). These approaches are typically based on dynamic exploration. We provide an alternative: a purely-static approach, which could produce more comprehensive models. Of course, dynamically-generated models can provide additional information that is not available statically—for example, whether certain widgets are disabled in a particular state. On the other hand, static analysis may expose behaviors that are only possible under complex run-time conditions that are unlikely to be triggered by automated dynamic exploration. The most natural direction to pursue is a hybrid static/dynamic approach, and existing work by Yang et al. [36] and Azim et al. [32] has already considered this possibility. With the help of information computed by static analysis (e.g., the events supported by a GUI widget, or the possible GUI transitions related to a widget), the dynamic analysis can be made more efficient and complete. Our static analysis is more general and comprehensive than the static analyses from [32], [36], and could potentially be integrated with their dynamic exploration.

VII. Conclusions

This works contributes to a growing foundation of static analyses for Android software. We develop a control-flow representation of user-driven callback behavior, using new context-sensitive analysis of event handlers. A client analysis for GUI model construction is also presented. Our experimental results highlight the promise of the proposed techniques. We believe that the CCFG (and its future generalizations) could be a valuable program representation for developing a variety of data-flow analyses for Android applications.

REFERENCES

[1] Gartner, Inc., "Worldwide traditional PC, tablet, ultramobile and mobile phone shipments," Mar. 2014, www.gartner.com/newsroom/id/2692318.

[2] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," in *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice Hall, 1981, pp. 189–234.

[3] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL*, 1995, pp. 49–61.

[4] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated security certification of Android applications," University of Maryland, College Park, Tech. Rep. CS-TR-4991, 2009.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *MobiSys*, 2011, pp. 239–252.

[6] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *NDSS*, 2012.

[7] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: An automatic system for revealing UI-based trigger conditions in Android applications," in *SPSM*, 2012, pp. 93–104.

[8] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. le Traon, "Effective inter-component communication mapping in Android with Epicc," in *USENIX Security*, 2013.

[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 *PLDI*, 2014, pp. 259–269.

[10] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?" in *MobiSys*, 2012, pp. 267–280.

[11] "Stopping and restarting an activity," developer.android.com/training/basics/activity-lifecycle/stopping.html.

[12] P. Dubroy, "Memory management for Android applications," in *Google I/O Developers Conference*, 2011.

[13] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *CCS*, 2012, pp. 229–240.

[14] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction," in *ICSE*, 2014, pp. 1036–1046.

[15] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *FSE*, 2014, pp. 576–587.

[16] S. Zhang, H. Lü, and M. D. Ernst, "Finding errors in multithreaded GUI applications," in *ISSTA*, 2012, pp. 243–253.

[17] E. Payet and F. Spoto, "Static analysis of Android programs," *IST*, vol. 54, no. 11, pp. 1192–1201, 2012.

[18] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based GUI testing of an Android application," in *ICST*, 2011, pp. 377–386.

[19] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *ISSTA*, 2013, pp. 67–77.

[20] D. Yan, S. Yang, and A. Rountev, "Systematic testing for resource leaks in Android applications," in *ISSRE*, 2013, pp. 411–420.

[21] S. Yang, D. Yan, and A. Rountev, "Testing for poor responsiveness in Android applications," in *MOBS*, 2013, pp. 1–6.

[22] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *ASE*, 2012, pp. 258–261.

[23] "GATOR: Program Analysis Toolkit For Android," web.cse.ohio-state.edu/presto/software/gator.

[24] "OpenManager: File manager for Android," github.com/nexes/Android-File-Manager.

[25] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *TOPLAS*, vol. 23, no. 6, pp. 685–746, 2001.

[26] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *TOSEM*, vol. 14, no. 1, pp. 1–41, 2005.

[27] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *POPL*, 2011, pp. 17–30.

[28] A. Rountev and D. Yan, "Static reference analysis for GUI objects in Android software," in *CGO*, 2014, pp. 143–153.

[29] D. Yan, "Program analyses for understanding the behavior and performance of traditional and mobile object-oriented software," Ph.D. dissertation, Ohio State University, Jul. 2014.

[30] A. Rountev, S. Kagan, and T. Marlowe, "Interprocedural dataflow analysis in the presence of large libraries," in *CC*, 2006, pp. 2–16.

[31] "SCanDroid: Security Certifier for anDroid," spruce.cs.ucr.edu/SCanDroid/tutorial.html.

[32] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *OOPSLA*, 2013, pp. 641–660.

[33] "Intents and intent filters," developer.android.com/guide/components/intents-filters.html.

[34] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *TCS*, vol. 167, no. 1-2, pp. 131–170, 1996.

[35] A. M. Memon, "An event-flow model of GUI-based applications for testing," *STVR*, vol. 17, no. 3, pp. 137–157, 2007.

[36] W. Yang, M. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *FASE*, 2013, pp. 250–265.

[37] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," in *ICSE*, 2012, pp. 595–605.

[38] M. Prasad, "Personal communication," 2014.

[39] P. Tramontana, "Android GUI Ripper," wpage.unina.it/ptramont/GUIRipperWiki.htm.

[40] 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 *SPSM*, 2013, pp. 21–32.

[41] E. Payet and F. Spoto, "An operational semantics for Android activities," in *PEPM*, 2014, pp. 121–132.

[42] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *WCRE*, 2003, pp. 260–269.

[43] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *TSE*, vol. 31, no. 10, pp. 884–896, 2005.

[44] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: High coverage, no false alarms," in *ISSTA*, 2012, pp. 67–77.

[45] P. Wang, B. Liang, W. You, J. Li, and W. Shi, "Automatic Android GUI traversal with high coverage," in *CSNT*, 2014, pp. 1161 – 1166.