Static Analyses of GUI Behavior in Android Applications

Dissertation

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Shengqian Yang
Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Dissertation Committee:

Atanas Rountev, Advisor

Michael D. Bond

Neelam Soundarajan

ABSTRACT

With the fast growing complexity of software systems, developers experience new challenges in understanding program's behavior to reveal performance and functional deficiencies and to support development, testing, debugging, optimization, and maintenance. These issues are especially important to mobile software due to limited computing resources on mobile devices, as well as short development life cycles. The correctness, security, and performance of mobile software is of paramount importance for many millions of users. For software engineering researchers, this raises high expectations for developing a comprehensive toolset of approaches for understanding, testing, checking, and verification of Android software. Static program analyses are essential components of such a toolset. Because of the event-driven and framework-based nature of the Android programming model, it is challenging to clearly understand application semantics and to represent it in static analysis algorithms. This dissertation makes several contributions towards solving this challenge.

The ability to understand the interprocedural control flow is critical for reasoning statically about the semantics of a program. For Android, this flow is driven by the Graphical User Interface (GUI) of the application. As the first contribution of this dissertation, we propose a novel technique that analyzes the control flow of GUI event handlers in Android software. We build a callback control-flow graph, using a context-sensitive static analysis of callback methods such as GUI event handlers. The algorithm performs a graph reachability analysis 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.

The next contribution of this dissertation is an even more general model and static analysis of the control flow of an Android application's GUI. We propose the window transition graph (WTG), a model representing the possible GUI window sequences and their associated events and callbacks. A key component and contribution of our work is the careful modeling of the stack of currently-active windows, the changes to this stack, and the effects of callbacks related to these changes. To the best of our knowledge, this is the first detailed study of this important static analysis problem for Android. We develop novel analysis algorithms for WTG construction and traversal, based on this modeling of the window stack. We also describe an application of the WTG for GUI test generation, using path traversals. The evaluation of the proposed algorithms indicates their effectiveness and practicality.

User's interactions with Android applications trigger callbacks in the UI thread. The handling of such events may initialize work on the background in order to perform expensive tasks. Because Android does not allow non-UI threads modifying the GUI state, standard Android "post" operations play a critical role in communicating between background and UI threads. To understand this additional aspect of Android semantics, we introduce a static analysis to model operations that post runnable tasks from non-UI threads to the UI thread's event queue. The results of this analysis are used to create a more general version of the WTG. This new WTG and the related static analysis present an important step toward other more comprehensive modeling of Android semantics. The experimental evaluation of the proposed representation indicates promising overall accuracy improvements.

To conclude, this dissertation presents several static analysis techniques to model the behaviors of the GUIs of Android applications. These analyses present essential foundation for developing tools to uncover the symptoms of both functional and performance issues in the mobile system, to perform model-based testing, and to support the understanding, optimization, and evolution of Android software.

To my family

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my advisor Prof. Atanas (Nasko) Rountev, for his support, guidance and patience during my Ph.D. study. I appreciate his dedication in training and helping me in the area of research. I would also like to thank Prof. Michael D. Bond and Prof. Neelam Soundarajan for serving on the dissertation committee. I must also acknowledge all the current and former PRESTO members for their insightful discussions and collaborations. A very special thanks to Igor Murashkin and Jean-Philippe Lesot for their mentoring on the intern project at Google. Last but not least, I would like to thank my family: my parents and my wife for their unconditional support and understanding through my good time and bad.

The material presented in this dissertation is based upon work supported by the U.S. National Science Foundation under grants CCF-1017204, CCF-1319695, and CCF-1526459, and by a Google Faculty Research Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VITA

September 2010 – August 2015	Graduate Teaching/Research Associate, The Ohio State University
May 2013	M.S. Computer Science, The Ohio State University
June 2010	B.Eng. Software Engineering, Shanghai Jiao Tong University

PUBLICATIONS

Research Publications

Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, Atanas Rountev. Static Window Transition Graphs for Android. In *International Conference on Automated Software Engineering (ASE'15)*, November 2015.

Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, Atanas Rountev. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *International Conference on Software Engineering (ICSE'15)*, May 2015.

Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. LeakChecker: Practical Static Memory Leak Detection for Managed Languages. In *International Symposium on Code Generation and Optimization (CGO'14)*, pages 87-97, February 2014.

Dacong Yan, Shengqian Yang, and Atanas Rountev. Systematic Testing for Resource Leaks in Android Applications. In *IEEE International Symposium on Software Reliability Engineering (ISSRE'13)*, pages 411-420, November 2013.

Shengqian Yang, Dacong Yan, and Atanas Rountev. Testing for Poor Responsiveness in Android Applications. In *International Workshop on the Engineering of Mobile-Enabled Systems (MOBS'13)*, pages 1-6, May 2013.

Shengqian Yang, Dacong Yan, Guoqing Xu, and Atanas Rountev. Dynamic Analysis of Inefficiently-Used Containers. In *International Workshop on Dynamic Analysis (WODA'12)*, pages 30-35, July 2012.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Programming Language	Prof. Atanas Rountev
Software Systems	Prof. Feng Qin
Software Systems	Prof. Srinivasan Parthasarathy

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: 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 [14].

This explosive growth in the number of deployed smartphones and tablets has significantly changed the computing landscape. The correctness, security, and performance of such devices is of paramount importance for many millions of users. For software engineering researchers, this raises high expectations for developing a comprehensive toolset of approaches for understanding, testing, checking, and verification of Android software. Static program analyses are essential components of such a toolset. In this work we focus on several such static analyses.

1.1 Challenges

Increasing complexity of mobile software. With increasing hardware performance, e.g., larger RAM, SSD, and powerful CPUs, it becomes easier to develop mobile software offering complicated functionality. However, the increasingly complex behavior of such software makes it difficult for developers to understand, test, and maintain the software systems. This situation is getting even worse due to the fact that the development life cycles for most mobile applications are short, in order to attract more users. For static analysis researchers that aim to develop software tools, this complexity creates a number of technical challenges. For example, for Android applications—which are driven by graphical user interfaces (GUIs)—the automated generation of a static GUI representation is an essential component facilitating software understanding and testing. While there are some techniques [7, 53] proposed to explore Android GUIs dynamically, there is little work on capturing the complex GUI structure and behavior statically. At present, it is difficult to fully rely on dynamic approaches to cover sufficient features in GUI-based Android software for the purposes of understanding and testing. Furthermore, such dynamic approaches cannot be used to check fully the absence of certain categories of defects (e.g., related to flow of sensitive information, or leaking of resources/energy).

Framework-based and event-driven mobile software. In Android, a rich variety of callbacks are defined by the Android framework model for events such as component creation/termination, user actions, device state changes, etc. Users primarily rely on gestures, e.g., click, swipe, etc., to communicate with applications. In the typical scenario, when an user action is issued, the Android OS will capture it, and send all the corresponding information, e.g., event type, screen pixel and etc., to the framework. Based on the received data, the framework will determine and trigger the associated callbacks. The application code and the Android platform interact through *callbacks*: calls from the platform's event processing code to the relevant methods defined in the application code. Depending on the application logic, callbacks could also be used to initialize the interactions among different Android components, e.g., open a new GUI window. Because of such component-based and event-driven nature, traditional static control-flow analysis cannot be directly applied to Android applications. How to precisely infer and represent the control flow of Android applications presents new challenges for static analysis of Android software.

Inadequate static analysis of Android GUI control flow. The representation and analysis of callback methods play a critical role in a static analysis for GUI model construction, and more generally, control/data-flow analysis for Android GUIs. Transitions between GUI windows are triggered by such methods, and during these transitions additional callbacks occur. However, the current state of the art in static analysis for Android is inadequate when it comes to represent such run-time behavior. For example, we have seen various cases from real applications where an event handler may force the closing of the current window and its predecessor window, while at the same time opening a new window; this leads to complicated interleavings of callbacks for these three windows. As another example, we have also seen many cases where the return from a window does not come back to the predecessor, but rather to another window displayed earlier. In existing work, there is no conceptual clarity on these possible run-time behaviors and how they can be analyzed in a static control-flow analysis.

1.2 Static Analyses of GUI Behavior in Android Applications

Our focus is on a key component of a static analysis toolset for Android: an analysis to create a model of an application's graphical user interface (GUI). Such a model can be used for program understanding, testing, and dynamic exploration [3, 7, 24, 52, 58, 59]. It could also potentially be a starting point for static data-flow analyses, for example, for checking of security properties, leak defects, and other correctness properties [6, 8–10, 12, 13, 16, 19, 22, 26–28, 36, 38, 39, 51, 64, 65].

1.2.1 Static Control-Flow Analysis of User-Driven Callbacks

The first contribution of this dissertation is a control-flow analysis that focuses on the lifecycle and interactions of *user-event-driven application components*. Our first goal is to formulate this control-flow analysis problem in terms of the traditional concepts of interprocedural control-flow analysis [42,48], thus to provide 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. Some existing work considers aspects of this problem. For example, FlowDroid [6] uses a static analysis that represents the possible orderings of lifecycle/event callbacks for a single activity, but there is no modeling of sequences involving multiple activities. The SCanDroid tool [13], which aims to model the sequence of callbacks to event handlers [47], exhibits similar lack of generality. Other related approaches (e.g., [25,28,65]) are also limited, as they make overly-conservative or overly-restrictive assumptions about sequencing constraints for callback invocation. Other related work attempts to resolve statically Android mechanisms for inter-component communication (e.g., [7, 9, 12, 13, 22, 35, 36]), but this information by itself is not enough to infer the possible run-time orderings of callbacks.

We propose the *callback control-flow graph* (CCFG), a static representation of possible callback sequences. 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. We consider one possible client analysis: 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., [24, 52, 58, 59]), and possible sequences of user GUI events can be derived from valid paths in the model. Prior work (e.g., [3, 7, 53, 54, 62]) constructs such models using dynamic analysis: the run-time state of the application's GUI is examined to determine possible GUI actions, these actions are triggered, and the resulting GUI changes are recorded. We propose an alternative: a purely-static approach, which could (and does) produce more comprehensive models.

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 [3]. This comparison indicates high static analysis precision and better coverage than what is achieved with dynamic analysis.

A 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.

Chapter 2 describes our control-flow analysis algorithm and its experimental evaluation. A description of this work also appeared in [60].

1.2.2 Static Window Transition Graphs

As described above, GUI models can be derived from dynamic analysis [3,7,53,54, 62]; alternatively, models with potentially-higher coverage can be derived statically from the CCFG. These static GUI models can be used, for example, to determine the possible sequences of events and window transitions in an Android application. However, the models derived from the CCFG lack generality, in that they (1) do not model the general effects of callbacks, including certain operations that close windows, (2) do not represent the interleavings of callbacks from multiple windows, (3) do not model the complicated semantics of pressing the hardware BACK button, and (4) do not capture system events (e.g., screen rotation) that significantly change the GUI state.

Neither the work from Chapter 2, nor earlier work on static control-flow analysis for Android [6, 7, 9, 12, 13, 13, 22, 25, 28, 35, 36, 47, 65], provide a comprehensive answer to the following fundamental question: *What are the possible sequences of events, window transitions, and callbacks?* The deficiencies in this prior work are due to the complex run-time semantics of Android, in terms of both structural elements (e.g., GUI windows, widgets, and event handlers) and their behavior (e.g., interactions between windows and their effects on callbacks).

To provide a more comprehensive modeling of important aspects of this run-time semantics, we propose a new GUI control-flow representation for Android: the *window transition graph* (WTG). Nodes in this graph represent windows and edges represent transitions between windows, triggered by callbacks executed in the UI thread. To allow the development of client data-flow analyses based on the WTG, graph edges are annotated with the sequences of callback methods invoked by the Android platform. These annotations capture event handling callbacks and window lifecycle callbacks. These callbacks are a strict superset of the ones included in the CCFG (which as outlined in Section 1.2.1). The WTG subsumes the CCFG, in the sense that it captures all information from the CCFG as well as additional aspects of possible behaviors. The static analysis to construct the WTG employs, as an important building block, the context-sensitive analysis of a callback method described in Section 1.2.1.

The GUI transitions in the WTG are represented with the help of the abstraction of a *window stack*. The window stack generalizes the standard Android notion of a "back stack" [2], which stores the currently-alive activities. (Activities correspond to one category of windows.) No prior work on static control-flow analysis for Android models this critical aspect of the run-time GUI behavior. Our generalization (1) captures additional categories of windows, and (2) models the changes to the window stack. An important observation is that a single transition in the WTG can have complex effects on the window stack: for example, it can pop and/or push windows, all as part of the same WTG edge. A major contribution of our work is the careful modeling of these stack changes and their related callbacks—both the callbacks that trigger the stack changes, and the callbacks triggered by them. To the best of our knowledge, this is the first detailed study of this important static analysis problem for Android.

The combined analysis of callbacks and the window stack also provides a solution to an important related problem: which sequences of window transitions are feasible?

One cannot consider all WTG paths, since some such paths are provably infeasible. We can draw an analogy with the sequences of calls/returns in ordinary programs: modeling the possible states of the call stack is a key concern in static analysis of call/return sequences (which, in turn, is an important component of interprocedural data-flow analysis). However, the behavior of the window stack can be significantly more complicated. Our work provides a systematic identification of *valid* WTG paths (and, by trivial extension, valid call/return sequences), which is a critical prerequisite for future developments in interprocedural data-flow analysis for Android. As an exemplar client, we have developed a test generation tool in which valid WTG paths naturally correspond to test cases.

The experimental evaluation indicates that event handlers can have complex behaviors and their transitions depend on non-trivial sequences of preceding events. Our WTG representation and static analysis are the first to model these features, leading to improved static GUI models and test case generation. For six applications, manual comparison with run-time behavior indicates that the analysis achieves good precision. In addition, our results indicate that the analysis running times are suitable for practical use.

Chapter 3 describes the WTG, the static analysis to construct it, and the related experimental evaluation. A description of this work will also appear in [61].

1.2.3 Modeling of Asynchronous Control Flow

The window transition graph represents only the control flow triggered by code executed in the main (UI) thread of the application. However, Android applications often offload work to separate threads, and these threads asynchronously trigger GUI changes by posting code to be executed by the UI thread. The last contribution of this dissertation is an extension of the WTG to represent common cases of such behavior, as well as the necessary modifications to the static analyses for WTG construction. The WTG is augmented with transition edges that represent the execution of such code (i.e., code posted to the UI thread) and static analysis of this code determines the effects of this execution. We present several case studies to demonstrate that the resulting WTG is more comprehensive than the one outlined earlier, and that the new WTG edges precisely represent the run-time behavior. Chapter 4 describes this approach and its experimental evaluation.

1.3 Outline

The rest of this dissertation is organized as follows. Chapters 2–4 present the novel program analysis techniques contributed by this dissertation. Related work is described in Chapter 5. Chapter 6 summarizes this dissertation's contributions and conclusions.

CHAPTER 2: Static Control-Flow Analysis of User-Driven Callbacks in Android Applications

2.1 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 a key component of data-flow analysis.

2.1.1 Background

The standard program representation for control flow 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) [42, 48].

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. These components execute in the application's UI thread (which is its main thread). 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 [10, 51, 58]). 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. Furthermore, we focus only on GUI changes triggered by user events; Chapter 4 describes some generalizations to handle certain GUI changes triggered by other threads from the same application.

2.1.2 Example from an Android Application

Figure 2.1 shows a simplified example derived from OpenManager [37], 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

```
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 = {
7        R.id.info_button, R.id.help_button,
8        R.id.manage_button, R.id.multiselect_button};
9      for(int i = 0; i < img_button_id.length; i++) {
10       ImageButton b = (ImageButton)findViewById(img_button_id[i]);
11       b.setOnClickListener(mHandler);
12     }
13   }
14 }
15 public class EventHandler implements 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(mActivity, DirectoryInfo.class);
24         mActivity.startActivity(info);
25         break;
26       case R.id.help_button:
27         Intent help = new Intent(mActivity, HelpManager.class);
28         mActivity.startActivity(help);
29         break;
30       case R.id.manage_button:
31         AlertDialog.Builder builder = ...
32         AlertDialog dialog = builder.create();
33         dialog.show();
34         break;
35       default:
36         ...
37         break;
38     }
39   }
40 }
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>
```

Figure 2.1: Example derived from OpenManager [37]

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` is the info button, line 24) or of `HelpManager` (when `v` is the help button, line 28). 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 33. 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 36); 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., [18, 33, 49]), 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`) [18]. 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.

2.1.3 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`).

We assume that relevant static abstractions have already been defined by an existing analysis of GUI-related objects [45, 56]. We will refer to this analysis as GATOR, using the name of its public implementation [15]. 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 2.1). 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 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 2.2. 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 [44]; recent work [6] also uses a similar approach, as discussed later. Both `onDestroy` methods are successors of

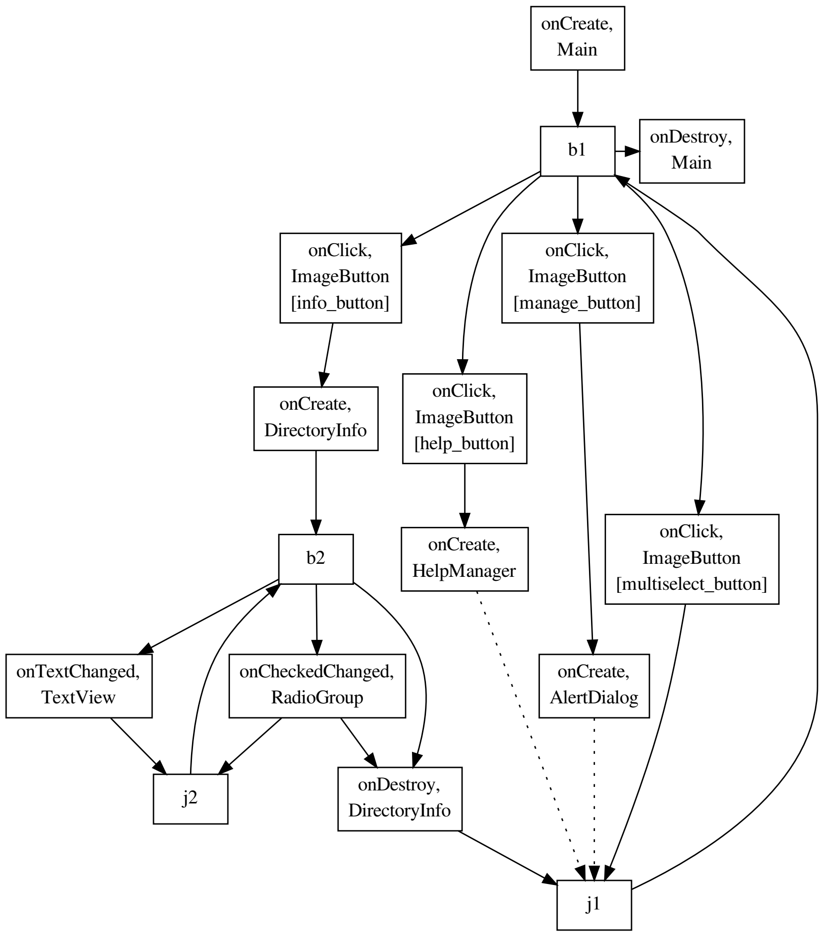

onCreate, Main
b1
onDestroy, Main
onClick, ImageButton [info_button]
onClick, ImageButton [manage_button]
onCreate, DirectoryInfo
onClick, ImageButton [help_button]
onClick, ImageButton [multiselect_button]
b2
onCreate, HelpManager
onTextChanged, TextView
onCheckedChanged, RadioGroup
onCreate, AlertDialog
j2
onDestroy, DirectoryInfo
j1

Figure 2.2: Callback control-flow graph

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 could be added for standard Android events such as screen rotation, interruption due to a phone call, or locking/unlocking the device screen [56], but we do not consider them in the work described in this chapter. The generalizations described in the next chapter capture the effects of such standard events.

2.1.4 Prior Work

Existing work has addressed some aspects of this problem. For example, FlowDroid [6] 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 [13], which aims to model the sequence of callbacks to event handlers [47], has similar limitations. Chapter 5 contains discussion of other related work.

Another area of related work is the resolution of activity-launch calls, such as the `startActivity` calls at lines 24 and 28 in Figure 2.1. Activity-launch APIs use an intent object to specify the target activity; two examples are shown at lines 23 and 27 in the figure. There are several existing techniques [7, 9, 12, 13, 22, 36] for analysis of intent objects. By itself, intent analysis cannot determine the edges in a CCFG (shown in Figure 2.2). 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 33), and (3) modeling of window termination calls. One component of our control-flow analysis is an intent analysis which is derived from prior work [36]. The details of this algorithm will be discussed in Section 2.2.4.

2.2 Analysis Algorithm

2.2.1 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 2.1. This algorithm is then used by the main control-flow analysis, as described in Section 2.2.2.

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 24, 28, and 33 in Figure 2.1; other examples are provided in Section 2.2.2. An analysis output is the set *reachedTriggers* of trigger nodes encountered during the traversal.

Algorithm 2.1: AnalyzeCallbackMethod(m,c)

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

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 2.1), 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 2.2.3. 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. Algorithm 2.1 is based on the general graph-traversal technique for solving interprocedural, finite, distributive, subset (IFDS) data-flow analysis problems [42]. We formulated an IFDS problem with a lattice containing two elements: $\emptyset$ and the singleton set $\{\mathit{entryNode}(m)\}$. 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 2.1.

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 [42].

2.2.2 CCFG Construction

CCFG construction uses the output from GATOR. In this output, an activity a is associated with widgets $\mathit{Views}(a) \subseteq \mathcal{V}$. 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 $\mathit{Views}(m)$, 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 $\mathit{Views}(d)$. A widget v could be associated with several event handlers $\mathit{Handlers}(v) \subseteq \mathcal{H}$.

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 $h \in \mathit{Handlers}(v)$ where $v \in \mathit{Views}(w)$, 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 2.2 defines the edges created for a window w. As illustrated in Figure 2.2, 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 2.2. The second edge represents the termination of w with the BACK button.

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 2.2, 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 2.2, created by line 12 in Algorithm 2.2.

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 2.2). 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 2.1 (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

Algorithm 2.2: 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  $newEdges \leftarrow \emptyset$
 2  $\langle triggers, avoids \rangle \leftarrow$ AnalyzeCallbackMethod$(l^c, w)$
 3  $newEdges \leftarrow newEdges \cup$ TriggerEdges$(triggers, l^c, w)$
 4  if $avoids$ then
 5      $newEdges \leftarrow newEdges \cup \{(l^c, w) \rightarrow b_w\}$
 6  $newEdges \leftarrow newEdges \cup \{b_w \rightarrow (l^t, w)\}$
 7  foreach event handler node $(h, v)$ do
 8      $newEdges \leftarrow newEdges \cup \{b_w \rightarrow (h, v)\}$
 9      $\langle triggers, avoids \rangle \leftarrow$ AnalyzeCallbackMethod$(h, v)$
10      $newEdges \leftarrow newEdges \cup$ TriggerEdges$(triggers, h, v)$
11      if $avoids$ then
12          $newEdges \leftarrow newEdges \cup \{(h, v) \rightarrow j_w\}$
13  if $w$ is not a menu then
14      $newEdges \leftarrow newEdges \cup \{j_w \rightarrow b_w\}$
15  else
16      $newEdges \leftarrow newEdges \cup \{j_w \rightarrow (l^t, w)\}$
```

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 2.1 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 24 and 28 in Figure 2.1) are analyzed with our flow- and context-insensitive intent analysis, conceptually derived from a more expensive prior analysis [36], but accounting for statement feasibility (analogous to line 1 in Algorithm 2.1). The analysis focuses on explicit intents because they are designed for use inside the same application [23]. In our experience, it performed as well as existing alternatives [36, 47]. Section 2.2.4 provides more details on this analysis.

Menu-launch calls such as `showContextMenu` as well as dialog-launch calls (e.g., line 33 in Figure 2.1), are resolved by GATOR. Any such statement triggers the launch of a new window w'. Correspondingly, function TRIGGEREDGES produces edges $(h, v) \to (l'^c, w')$ and $(l'^t, w') \to j_w$ when invoked at line 10, and edges $(l^c, w) \to (l'^c, w')$ and $(l'^t, w') \to b_w$ 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 $(h, v) \to (l^t, w)$ or $(l^c, w) \to (l^t, w)$ to represent the possible flow of control.

Example. For the running example shown in Figure 2.1, calling ANALYZECALLBACKMETHOD at line 9 with h = `onClick` and v = `ImageButton[info_button]` will return *triggers* = $\{s_{24}\}$ and *avoids* = *false*. Activity-launch statement s_{24}, representing line 24 in Figure 2.1, is resolved to w' = `DirectoryInfo`. As a result, edges $(h, v) \to$ (`onCreate`, w') and (`onDestroy`, w') $\to j_1$ 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.

2.2.3 Detection of Feasible Edges

CCFG construction depends on constant propagation analysis to determine the feasible edges under a particular context (recall the invocation of ComputeFeasibleEdges at line 1 of Algorithm 2.1). 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 2.1, 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 [46].

2.2.4 Intent Analysis

In Android, inter-component communication is based on *intents*. Such objects are used, for example, in activity-launch calls such as `startActivity` (e.g., lines 24 and 28 in Figure 2.1). The intent objects are defined with or without explicit target components. An explicit intent specifies the targeted component, while an implicit intent specifies other information that can be used to indirectly infer the targeted component(s). In general, explicit intents are used for intra-application interactions, that is, when the target component is inside the same application (e.g., another application activity). Implicit intents are mostly employed for inter-application communication. The intents shown in Figure 2.1 are explicit, as they specify the target components (activity `DirectoryInfo` or `HelpManager`) as parameters of the constructor calls at lines 23 and 27.

Our current analysis is integrated with an analysis of explicit intents, as we focus on analyzing intra-application transitions. Our prior work [45] builds a constraint graph tracking the flow of objects. This graph does not record the propagation of constant values that are used to specify the targets of explicit intents: specifically, class constants `X.class` (as shown at lines 23 and 27 in Figure 2.1) and string constants `"X"`.

To analyze intent objects and the targets they specify, the existing flow graph is extended to capture value propagation for such constants, and to model API calls that define the content of intent objects. Additional flow graph nodes and edges are introduced for assignments involving class constants and string constants, e.g., `x = X.class` and `x = "X"`. Because our implementation is based on the Soot static analysis framework [50], such assignments are easy to identify in the program representation. Flow graph nodes for the constant values are introduced, together with edges to the corresponding left-hand-side local variables. Another flow graph extension involves nodes to represent methods calls that modify the content of an intent object. The Android framework defines several such API calls. For example, method `setClass` of in class `Intent` sets the target through a provided `Class` argument. Another example is an `Intent` constructor for which the targeted component is defined by a string parameter or a class constant (e.g., at lines 23 and 27 in Figure 2.1). Besides these APIs, we also model methods which use an existing intent to copy all its data into another one, e.g., method `fillIn` could copy all data from its parameter intent to its receiver intent. A similar example is an `Intent` copy constuctor which takes as a parameter another intent.

Algorithm 2.3 resolves intent targets without considering the invocation context of a callback event handler. In addition to the extended flow graph, the algorithm takes as input the set *setIntentTargetMethods* of methods that could modify the target of an intent objects (e.g., `setClass`, `fillIn`, etc.) Helper function GETFLOWFROM

```
Algorithm 2.3: ResolveIntentContextInsensitive()
   Input: $fg$ : extended flow graph
   Input: $setIntentTargetMethods$ : methods to set the content of an intent
   Output: $resolvedIntents \leftarrow \emptyset$ : multimap from intent objects to targets
1  $worklist \leftarrow \emptyset$
2  $intentToSetContent \leftarrow \emptyset$
3  foreach $node\ n\ in\ fg$ do
4      if $n$ is intent allocation expression then
5          $toNodes \leftarrow \textsc{GetFlowFrom}(fg, n)$
6          foreach $node\ setIntentContent \in toNodes$ do
7              if $setIntentContent$ is invocation of
               $m \in setIntentTargetMethods \wedge setIntentContent$ modifies target of $n$ then
8                  if $m$ propagates all values from one intent to another intent then
9                      $worklist \leftarrow worklist \cup \{n \rightarrow setIntentContent\}$
10                 else
11                     $arg \leftarrow \textsc{GetIntentTargetArg}(setIntentContent)$
12                     foreach $tgt \in \textsc{GetFlowTo}(fg, arg) \wedge tgt$ is class/string constant do
13                         $resolvedIntents \leftarrow resolvedIntents \cup \{n \rightarrow tgt\}$
14                 $intentToSetContent \leftarrow intentToSetContent \cup \{n \rightarrow setIntentContent\}$
15 while $worklist \neq \emptyset$ do
16     $stable \leftarrow true$
17     $worklist \leftarrow worklist - \{intent_1 \rightarrow propagateAllContent\}$
18     $arg \leftarrow \textsc{GetIntentAllArg}(propagateAllContent)$
19     foreach $intent_2 \in \textsc{GetFlowTo}(fg, arg) \wedge intent_2$ is intent allocation expression do
20         foreach $tgt \in \textsc{GetIntentContent}(resolvedIntents, intent_2)$ do
21             if $tgt \notin \textsc{GetIntentContent}(resolvedIntents, intent_1)$ then
22                 $resolvedIntents \leftarrow resolvedIntents \cup \{intent_1 \rightarrow tgt\}$
23                 $stable \leftarrow false$
24     if $stable = false$ then
25         foreach $setIntentContent_1 \in \textsc{GetFlowFrom}(fg, intent_1)$ do
26             if $setIntentContent_1$ is invocation of $m \in setIntentTargetMethods \wedge m$
               propagates all values from one intent to another intent $\wedge\ intent_1$ is the source
               intent then
27                 foreach $\{intent_2 \rightarrow setIntentContent_2\} \in intentToSetContent$ do
28                     if $setIntentContent_1 = setIntentContent_2$ then
29                         $worklist \leftarrow worklist \cup \{intent_2 \rightarrow setIntentContent_2\}$
```

(or getFlowTo) traverses the flow graph from the given node to identify all nodes reachable from (or reaching) it.

The first part of the algorithm (line 3–14) performs an initial resolution of intents. For each node representing an intent allocation expression, (i.e., `new Intent`), the nodes setting its content are examined (line 7). If a node modifies the intent by passing another intent, the propagation relationship between the intent and the node will be remembered in a worklist (line 9), which is processed later by the second part of this algorithm. Otherwise, the local variable corresponding to the intent target will be retrieved by helper function GetIntentTargetArg. After retrieving the local variable *arg* used to set the target, all constants that flow to it are recorded as targets (line 13). The second part of this analysis (lines 15–29) is a worklist algorithm which incrementally updates the targets of resolved intents. Each element in the worklist is a pair of an intent $intent_1$ and a node that propagates all data of another intent $intent_2$ into $intent_1$. Similarly to GetIntentTargetArg, GetIntentAllArg finds the local variable *arg* used to provide $intent_2$. For example, statement `x.fillIn(y)` copies the data from the intent pointed-to by local variable `y` into the intent pointed-to by local variable `x`. To capture this flow, line 19 traverses the flow graph to find relevant $intent_2$ in order to update the targets of $intent_1$ (line 22). If a change is observed, any potentially-affected intents are added to the worklist to be processed in the future (lines 24–29).

Algorithm 2.3 resolves intent targets without considering the callback context. To improve the precision of intent analysis, our implementation refines this analysis by utilizing the outputs from method ComputeFeasibleEdges used in Algorithm 2.1. Given the callback and context, this method generates a set of reachable statements.

Only the flow graph nodes related to those statements are considered in the context-sensitive resolution. If this approach cannot resolve the target activity for a statement, the conservative solution generated by Algorithm 2.3 is used instead.

2.2.5 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 2.2, 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 $(\texttt{onClick}, \texttt{info\_button}) \rightarrow$ $(\texttt{onCreate}, w')$, 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 [18,48]. Note that the next chapter presents a generalized analysis of valid sequences of events and callbacks. That analysis is more general than the CCFG path analysis outlined above.

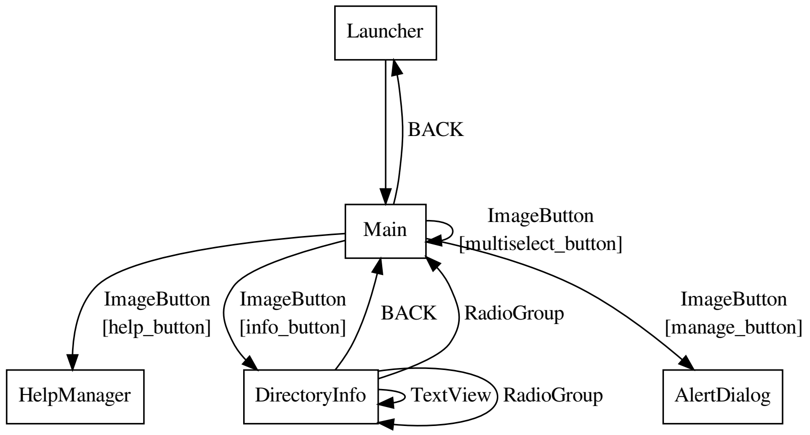


Figure 2.3: GUI model for the running example

2.3 Client Analysis: Construction of GUI Models

The control-flow analysis described above could potentially be used as a component of other static analyses (e.g., [6, 9, 10, 12, 13, 16, 22, 28, 36, 38, 39, 51, 64, 65]). 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., [3, 29, 52, 62]). Figure 2.3 shows an example of the GUI models we consider; they are similar in spirit to those used in prior work. The model is a directed graph in which 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 2.3. "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.

Such a model can serve as starting point for test generation. For example, an existing automated test generation technique for Android [24] 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 [58] 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 [52, 59], 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. 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 2.3) 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 [58]), 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.

2.4 Experimental Evaluation

We applied the analysis on 20 open-source Android applications used in prior work [39, 45, 58, 62, 63]. Our goals were to (1) characterize the size and complexity of the CCFG, (2) measure the benefits of context sensitivity in the analysis of event handlers, and (3) evaluate the precision of the GUI models derived from the CCFG.

2.4.1 CCFG Construction

Table 2.1(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 Table 2.2: 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. The ordering constraints for such callbacks are modeled by the output of our analysis.

Column "OutDegree" shows the average number of outgoing edges for CCFG nodes corresponding to event handlers. This number is an indication of the variability

Table 2.1: Characteristics of the analyzed applications and their CCFG analysis time.

(a) Applications						(b) Times	
Name	Classes	Methods	Activities	Menus	Dialogs	CS [s]	CI [s]
APV	68	413	4	4	5	10	8
Astrid	1228	5782	41	3	48	105	54
BarcodeScanner	126	594	9	4	6	10	10
Beem	284	1883	12	6	5	12	12
ConnectBot	371	2366	11	8	17	28	22
FBReader	954	5452	27	9	8	843	269
K9	815	5311	32	3	19	79	54
KeePassDroid	465	2784	20	11	9	27	20
Mileage	221	1223	50	15	9	13	12
MyTracks	485	2680	32	8	20	23	21
NPR	249	1359	13	12	6	16	15
NotePad	89	394	8	3	10	9	9
OpenManager	53	237	6	2	9	7	6
OpenSudoku	140	726	10	6	18	11	11
SipDroid	331	2863	12	5	13	39	34
SuperGenPass	64	267	2	2	4	8	8
TippyTipper	57	241	6	3	0	6	6
VLC	242	1374	10	2	13	19	18
VuDroid	69	385	3	2	1	5	5
XBMC	568	3012	22	20	24	38	32

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 2.1 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 Table 2.3, which contains similar measurements for

Table 2.2: Average out degree of CCFGs for applications in Table 2.1.

	CCFGs			
Name	Nodes	Edges	OutDegree	OutDegCI
APV	88	158	1.15	2.98
Astrid	980	1896	1.14	1.14
BarcodeScanner	104	171	1.37	1.88
Beem	121	186	1.14	2.20
ConnectBot	197	317	1.20	1.20
FBReader	272	2916	11.41	12.59
K9	393	723	1.15	1.59
KeePassDroid	288	682	2.01	2.47
Mileage	522	914	1.34	1.70
MyTracks	286	630	1.83	4.31
NPR	564	1175	1.19	2.08
NotePad	134	259	1.32	2.84
OpenManager	110	183	1.10	2.30
OpenSudoku	170	307	1.41	3.44
SipDroid	148	346	2.00	3.98
SuperGenPass	61	107	1.18	1.64
TippyTipper	61	94	1.00	1.24
VLC	169	278	1.10	1.10
VuDroid	35	62	1.50	3.33
XBMC	2275	6254	1.85	2.24

the average number of outgoing edges for a node in the static GUI model. (Back button edges were not included in these measurements, since in this GUI model 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 2.4.2 provides additional case studies on the benefits of context sensitivity for GUI models.

We investigated two programs with significantly higher out-degrees measurements in Table 2.3, compared to the rest of the programs: `FBReader` and `XBMC`. For `FBReader`,

Table 2.3: Average out degree of GUI models for applications in Table 2.1.

	Models	
Name	OutDegree	OutDegCI
`APV`	4.23	10.85
`Astrid`	12.32	12.36
`BarcodeScanner`	3.28	4.44
`Beem`	2.57	5.24
`ConnectBot`	3.20	3.20
`FBReader`	55.95	61.67
`K9`	5.90	7.96
`KeePassDroid`	12.82	17.18
`Mileage`	5.41	6.64
`MyTracks`	8.53	18.71
`NPR`	34.29	59.29
`NotePad`	5.48	10.38
`OpenManager`	4.31	9.06
`OpenSudoku`	3.12	7.26
`SipDroid`	8.47	15.10
`SuperGenPass`	5.00	6.88
`TippyTipper`	4.13	5.13
`VLC`	4.24	4.24
`VuDroid`	3.67	8.00
`XBMC`	176.07	186.33

the culprit is a utility method that is called by the handlers of about 37% of the CCFG nodes. These nodes have significantly higher out-degrees 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, 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 [45]: because the analysis is context-insensitive, there is spurious propagation of widgets.

Part (b) of the Table 2.1 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.

2.4.2 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: `APV`, `BarcodeScanner`, `OpenManager`, `SuperGenPass`, `TippyTipper`, and `VuDroid`. These applications have the smallest number of windows in Table 2.1, and were chosen to allow comprehensive manual examination.

We compared our static approach with Android GUI Ripper [3] ("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 [53] 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. This approach has the advantage of observing the full details of run-time state and behavior, but is inherently limited in its ability to find all feasible transitions. In our experiments we let Ripper run to completion; the running

[1]We initially also considered another dynamic exploration tool [7] but observed that sometimes it achieved lower GUI coverage. We also attempted to obtain the reverse engineering tool used in [62], but its proprietary implementation could not be distributed outside of the company [41].

Table 2.4: 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

times are shown in the last column of Table 2.4. 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 2.4. 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 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.

2.5 Summary

In this chapter 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 importance of context sensitivity in the design of the analysis algorithm, and indicate good precision and practical cost for the proposed techniques.

CHAPTER 3: Static Window Transition Graphs for Android

The GUI models derived from the CCFG lack generality, in that they (1) do not model the general effects of callbacks, including certain operations that close windows, (2) do not represent the interleavings of callbacks from multiple windows, (3) do not model the complicated semantics of pressing the hardware BACK button, and (4) do not capture system events (e.g., screen rotation) that significantly change the GUI state. To solve this problems, we propose a more general GUI control-flow representation for Android: the *window transition graph* (WTG).

3.1 Android Behavior and Its WTG Representation

3.1.1 Relevant Android Features

For ease of presentation, we discuss again, through an example, some of the major Android features relevant to our analysis. Figure 3.1 contains an example derived from the APV PDF viewer [5]. For simplicity, the code and its description omit a number of non-essential details. The example illustrates windows (e.g., `ChooseFileActivity`), GUI widgets (e.g., `fileListView`), and event handlers (e.g., `onItemClick`).

Windows. Subclasses of `android.app.Activity` are used to define activities, which are core application building blocks. `ChooseFileActivity`, `OpenFileActivity`, `Options`, and `About` from Figure 3.1 are such classes; execution starts from an instance of `ChooseFileActivity`, which shows a file list. An activity displays a window containing several GUI widgets. A widget (also referred to as a "view") is an instance of a *view class*. In Figure 3.1, variables that refer to widgets include `fileListView` (list of files), `l` (the same list), `item` (individual list element), `aboutItem`, `optionsItem` (both are elements of a menu, as described below), and `btn` (a button).

We also consider the two other common categories of Android windows: menus and dialogs. Instances of menu classes represent short-lived windows associated with activities ("options" menus) and widgets ("context" menus). In Figure 3.1 `OpenFileActitivy` has an options menu, initialized by `onCreateOptionsMenu` to contain menu items `aboutItem` and `optionsItem`. A dialog is an instance of a subclass of `android.app.Dialog`. Both menus and dialogs are used for modal events that require users to take an action before they can proceed [1].[2] We will use **Win** to denote the set of all run-time windows (activities, menus, and dialogs), and **View** for the set of all run-time widgets in these windows.

A menu/dialog takes control temporarily for a simple interaction with the user, and its lifetime is shorter than activity lifetime. The last activity that was displayed before a menu or a dialog was displayed is considered to be the *owner activity* of this menu/dialog. In the running example, the options menu is owned by `OpenFileActivity`. There are more general cases: for example, in `OpenFileActivity` there exists a button (not shown in Figure 3.1) for which a long-click event opens a context menu m_1, in which a menu item can be clicked to open a dialog d_1 asking for a page number in the PDF file; if an incorrect number is entered, d_1 shows another dialog d_2 with an error message. In this example `OpenFileActivity` is the owner activity of m_1, d_1, and d_2. The lifetime of a menu or a dialog is contained within the lifetime of its owner activity.

[2]Such windows are common: for example, in our experiments, more than half of window transitions involved menus and dialogs.

```
class ChooseFileActivity extends Activity
  implements onItemClickListener {
  ArrayList<FileListEntry> fileList;
  ListView fileListView;
  // === Lifecycle callbacks ===
  void onCreate() { ...
    fileListView.setOnItemClickListener(this); }
  // Other lifecycle callbacks: onDestroy, onStart,
  // onRestart, onStop, onResume, onPause
  // === Widget event handler callback ===
  void onItemClick(ListView l, View item, int p) {
    FileListEntry entry = fileList.get(p);
    File file = entry.getFile();
    if (!file.exists()) return;
    Intent in = new Intent(OpenFileActivity.class);
    // initialize intent based on file
    startActivity(in); } }
class OpenFileActivity extends Activity {
  MenuItem aboutItem, optionsItem;
  // === Lifecycle callbacks ===
  // onCreate, onDestroy, etc.
  void onCreateOptionsMenu(Menu menu) {
    aboutItem = menu.add("Item");
    optionsItem = menu.add("Options"); }
  void onOptionsMenuClosed(Menu menu) { ... }
  // === Widget event handler callback ===
  void onOptionsItemSelected(MenuItem item) {
    if (item == aboutItem)
      startActivity(new Intent(About.class));
    if (item == optionsItem) {
      startActivity(new Intent(Options.class));
      this.finish(); } }
class Options extends Activity
  implements OnClickListener {
  Button btn;
  void onCreate() { btn.setOnClickListener(this); }
  void onClick(View v) {
    startActivity(new Intent(About.class));
    this.finish(); } } }
class About extends Activity { ... }
```

Figure 3.1: Example derived from the APV PDF reader [5].

Events. Each $w \in \mathbf{Win}$ can respond to several events. *Widget events* are of the form $e = [v,t]$ where $v \in \mathbf{View}$ is a widget and t is an event type (e.g., v could be a button and t could be "click"). We also consider five kinds of *default events*. Event *back* corresponds to pressing the hardware BACK button, which typically (but not always) returns to the window that triggered the current window. Event *rotate* shows that the user rotates the screen, which triggers various GUI changes. For example, if the currently-active window is a dialog, this dialog is destroyed, its underlying activity is also destroyed, and the activity (but not the dialog) is recreated and redisplayed. Event *home* abstracts a scenario there the user switches to another application and then resumes the current application (e.g., by pressing the hardware HOME button to switch to the launcher, and then eventually returning to the application). Event *power* represents a scenario where the device is put in low-power state by pressing the hardware POWER button, followed by device reactivation. Event *menu* shows the pressing of the hardware MENU button to display an options menu (or a click to display the hidden parts of an action bar). A default event will be represented as $e = [w,t] \in \mathbf{Win} \times \{back, rotate, home, power, menu\}$ where w is the currently-active window. We will use **Event** to denote the set of all widget events and default events.

Callbacks. Each $e \in \mathbf{Event}$ triggers a sequence of callbacks that can be abstracted as $[o_1,c_1][o_2,c_2]\ldots[o_k,c_k]$. Here c_i is a callback method and o_i is a run-time object on which c_i was triggered. We focus on two categories of callbacks. *Widget event handler callbacks* respond to widget events. Figure 3.1 shows three examples. Method `onItemClick` handles click events for items of list `fileListView`. The call at line 7 registers the activity with a listener for such events. The list, the item being clicked, and its position in the list are provided as parameters to the callback. Method `onOptionsItemSelected` handles clicks for items in the options menu, and takes the clicked item as a parameter. Method `onClick` at lines 37–39 responds to clicks on `btn`.

Lifecycle callbacks are used for lifetime management of windows. These methods are of significant interest to developers (e.g., in order to avoid leaks [10, 19, 51, 58]). There are seven kinds of lifecycle callbacks for activities, as indicated in Figure 3.1.

For example, creation callback `onCreate` indicates the start of the activity's lifetime, and termination callback `onDestroy` indicated end of lifetime. Menus and dialogs can also have create/terminate callbacks, for example, `onCreateOptionsMenu` and `onOptionsMenuClosed` in Figure 3.1.[3] We will use abstract names *create* and *destroy* to represent these create/terminate callbacks. Similarly, $start, \ldots, pause$ will denote corresponding callbacks in activities and (if applicable) in dialogs and menus. Let **Cback** be the set of all lifecycle and widget event handler callbacks.

3.1.2 Motivation and Related Work

Section 3.1.3 describes the window transition graph, our proposed static representation of window transitions and callbacks. Each node corresponds to a window and each edge represents a window transition, labeled with a callback sequence. Figure 3.2 shows the WTG for the running example.

Why this static representation? A number of challenging software engineering problems for Android can be addressed with static analyses where the modeling of control flow plays a critical role. A few examples include checking of security properties (e.g., [6, 9, 12, 13, 16, 22, 28, 36, 65]), detection of energy defects (e.g., [8, 27, 38]), leak defects (e.g., [10, 19, 51, 58]), data races (e.g., [26]), and other correctness checking (e.g., [39, 64]). For example, common battery-drain defects—"no-sleep" [38] and "missing deactivation" [8, 27]—can be stated as properties of callback sequences. These sequences could potentially be derived from WTG paths. Prior work [38] defines a data-flow analysis to identify relevant API calls (e.g., GPS is turned on) and to search for no-sleep paths along which corresponding turn-off/release calls are

[3]There is a related callback `onPrepareOptionsMenu`; for simplicity, it is not discussed here, but our implementation handles it.

missing. For this work, the order of callbacks is of critical importance, but their solution lacks generality and precision, and may even involve manual efforts by the user. Some dynamic analyses of energy defects [8, 27] also consider paths in which a sensor (e.g., the GPS) is not put to sleep appropriately, often because of mismanagement of lifecycle callbacks. A static approach to identify such code paths requires callback ordering information, and the WTG can provide this information. Another example is static taint analysis for Android. Representative algorithms such as [6] do not model soundly all callback interleavings and do not employ the control-flow validity constraints captured in our work. Future work could investigate whether such analyses benefit from the WTG representation. Yet another example is static detection of resource leaks. Such leaks are often the result of improper resource management under event/callback sequences [10, 19, 51, 58], including events such as *rotate*, *home*, and *back*. Developing static leak detectors requires callback sequences, which could be obtained from the WTG.

In addition to defect detection, the WTG is directly applicable for *GUI model construction* for program understanding, testing [24, 52, 58, 59], and dynamic exploration [3, 7, 62]. In Section 3.3 we describe a test generation tool we developed based on the WTG, using traversals of valid WTG paths.

Related work. Despite the critical importance of analyzing statically the possible GUI behaviors of an Android application, the current state of the art lacks a systematic and comprehensive solution. For example, an activity transition graph is constructed in [7] to guide run-time GUI exploration, but the underlying static analysis [13, 47] uses conservative assumptions about GUI-related control flow, and does not model the changes to the window stack. Other work that creates static GUI models (e.g., [65]) also lacks generality and representations of the window stack. Our earlier work [60] considers analysis of callbacks and determines ordering constraints between them. However, it also does not provide a comprehensive solution: (1) it considers only a limited subset of lifecycle callbacks; (2) it does not represent the interleavings of callbacks from multiple windows, as illustrated in Table 3.1, 3.2; (3) it does not model the window stack (e.g., it assumes that each *back* event will return to the previous window); (4) it does not handle the owner-close operations described shortly; (5) it does not consider *rotate*, *home*, and *power* events. Other work that analyzes possible callbacks in Android (e.g., [6, 12, 22, 25, 26, 28]) has similar or even more significant limitations. To the best of our knowledge, the proposed static analysis is the first comprehensive solution to the important problem of modeling the possible window/callback sequences in an Android GUI.[4]

3.1.3 Modeling of Window Transitions

Opening and closing of windows. Each callback could open a new window or close an existing one. Consider the following scenario: when an "Exit" button in an activity a is clicked, the corresponding event handler opens a new dialog d to ask the user to confirm the exit. When the dialog's "Yes" button b is selected, its handler h closes both the dialog as well as its owner activity a, and control returns back to some prior activity a'. At each event, various callbacks occur. For example, clicking b triggers [b,h] [d,*destroy*] [a,*pause*] [a',*restart*] [a',*start*] [a',*resume*] [a,*stop*] [a,*destroy*]. Our goal is to model statically such behavior and the related changes to the window

[4]Since our approach is tightly coupled with Android-specific semantics, it is unlikely that it will be relevant beyond Android code.

stack.[5] Note that we focus on the behavior of the main thread (i.e., UI event thread) of the application; analysis of multiple threads (e.g., as done in [26]) or of control flow across applications is not being considered. Additional limitations of the approach are discussed in Section 3.2.4.

There are various API calls to open and close windows. For example, a call to `startActivity` opens a new activity, and a call to `finish` closes an existing one. Similarly, calls to `show` and `dismiss` can create and destroy a dialog. These will be represented with abstract operations $open(w)$ and $close(w)$, where w is the window being created/destroyed. We have never encountered an example of an execution of a callback method c that opens more than one window, and thus we assume that any path through c contains at most one $open(w)$ operation.

Operations $close(w)$ may also be triggered during an execution of c. The two common patterns are *self-close* and *owner-close*. In a self-close, c is associated with a window w and c's execution issues $close(w)$; an example is shown at line 39 of Figure 3.1. Another example is `onOptionsItemSelected` associated with the options menu m: the semantics of menu-item-click event handlers includes an implicit menu self-close operation $close(m)$ that does not appear in the code. In owner-close operations, if c is associated with a menu m or a dialog d, it may issue $close(a)$ for the owner activity a. For example, the path at lines 30–32 in Figure 3.1 has an open operation followed by owner-close at line 32 and then an implicit self-close.

[5]The discussion assumes Android version 4.3; some earlier versions have slight variations in certain sequences of callbacks.

Note that the actual opening/closing of windows, as well as the related lifecycle callbacks, happen only after the callback issuing the open/close operations has completed. For example, after lines 30–32 are executed and `onOptionsItemSelected` completes, menu m and its owner $a =$ `OpenFileActivity` are closed, activity $a' =$ `Options` is opened, and the following callbacks are observed: [m,*destroy*] [a,*pause*] [a',*create*] [a',*start*] [a',*resume*] [a,*stop*] [a,*destroy*]. The ordering of open and close operations in a callback's execution path typically does not affect the outcome of its execution.

Behavior of the window stack. The window stack represents the set of currently-alive windows. The window that currently interacts with the user is on top of the stack. Due to space limitations, we describe the case where open and close operations appear only in widget event handler callbacks. Our algorithms and implementation also handle common cases where such operations occasionally appear in lifecycle callbacks.

The window stack starts a single element: the starting activity a. The creation of this initial state is associated with the lifecycle callback sequence to initialize a: [*create*,a] [*start*,a] [*resume*,a]. At any moment of time, the window $w \in \mathbf{Win}$ at the top of the stack determines the possible events that could be triggered by the user. These include widget events [v,t] where $v \in \mathbf{View}$ is a widget defined by w and t is the event type, as well as default events such as [w, *back*], etc. When a widget event [v,t] is triggered, callback [v,h] is invoked. Here $h \in \mathbf{Cback}$ is the corresponding event handling method, invoked on that same widget v. If h triggers a self-close

Table 3.1: Some window stack changes and callback sequences (part 1).

	Stack	Event	Handler	Open/Close	Stack changes
1	$(\ldots, a)$	$[v,t]$	$[v,h]$	none	none
2	$(\ldots, a)$	$[v,t]$	$[v,h]$	$open(a')$	push a'
3	$(\ldots, a', a)$	$[v,t]$	$[v,h]$	$close(a)$	pop a
4	$(\ldots, a)$	$[v,t]$	$[v,h]$	$close(a)$, $open(a')$	pop a, push a'
5	$(\ldots, a', a)$	$[a,back]$	implicit	$close(a)$	pop a
6	$(\ldots, a)$	$[a,rotate]$	implicit	$close(a)$, $open(a)$	pop a, push a
7	$(\ldots, a)$	$[a,home]$	implicit	none	none
8	$(\ldots, a)$	$[a,power]$	implicit	none	none
9	$(\ldots, a)$	$[a,menu]$	implicit	$open(m)$	push m
10	$(\ldots, a)$	$[v,t]$	$[v,h]$	$open(m)$	push m
11	$(\ldots, a)$	$[v,t]$	$[v,h]$	$open(d)$	push d
12	$(\ldots, a, m)$	$[v,t]$	$[v,h]$	$close(m)$	pop m
13	$(\ldots, a, m)$	$[v,t]$	$[v,h]$	$close(m)$, $open(a')$	pop m, push a'
14	$(\ldots, a', a, m)$	$[v,t]$	$[v,h]$	$close(m)$, $close(a)$	pop m, pop a
15	$(\ldots, a, m)$	$[v,t]$	$[v,h]$	$close(m\&a)$, $open(a')$	pop $m\&a$, push a'
16	$(\ldots, a, m)$	$[m,back]$	implicit	$close(m)$	pop m
17	$(\ldots, a, m)$	$[m,rotate]$	implicit	$close(m\&a)$, $open(a\&m)$	pop $m\&a$, push $a\&m$
18	$(\ldots, a, m)$	$[m,home]$	implicit	$close(m)$	pop m
19	$(\ldots, a, d)$	$[v,t]$	$[v,h]$	$open(a')$	push a'

operation, w is popped from the window stack. If, in addition, h triggers an owner-close operation, the owner activity is also popped from the top of the stack. [6] Finally, if h opens a new window, this window is pushed on top of the stack.

Some of these scenarios are summarized in Table 3.1 and Table 3.2. The first column describes the stack state, with the currently-visible window on top. We use a and a' to denote activities, m to denote an options menu, and d denote a dialog. Only a representative sample of cases are described; additional details on the scenarios captured by our algorithm are presented in Table 3.3 and Table 3.4. In several rows

[6]Since the lifetime of a menu/dialog is contained within the lifetime of its owner, closing an owner implies that all owned windows have been closed.

Table 3.2: Some window stack changes and callback sequences (part 2).

	Callback sequence
1	$[v,h]$
2	$[v,h][a,pause][a',create][a',start][a',resume][a,stop]$
3	$[v,h][a,pause][a',restart][a',start][a',resume][a,stop][a,destroy]$
4	$[v,h][a,pause][a',create][a',start][a',resume][a,stop][a,destroy]$
5	$[a,pause][a',restart][a',start][a',resume][a,stop][a,destroy]$
6	$[a,pause][a,stop][a,destroy][a,create][a,start][a,resume]$
7	$[a,pause][a,stop][a,restart][a,start][a,resume]$
8	$[a,pause][a,stop][a,restart][a,start][a,resume]$
9	$[m,create]$
10	$[v,h][m,create]$
11	$[v,h][d,create]$
12	$[v,h][m,destroy]$
13	$[v,h][m,destroy][a,pause][a',create][a',start][a',resume][a,stop]$
14	$[v,h][m,destroy][a,pause][a',restart][a',start][a',resume][a,stop][a,destroy]$
15	$[v,h][m,destroy][a,pause][a',create][a',start][a',resume][a,stop][a,destroy]$
16	$[m,destroy]$
17	$[a,pause][m,destroy][a,stop][a,destroy][a,create][a,start][a,resume][m,create]$
18	$[a,pause][m,destroy][a,stop][a,restart][a,start][a,resume]$
19	$[v,h][a,pause][a',create][a',start][a',resume][a,stop]$

the event handler is listed as “implicit”, because it is defined by the Android platform semantics and not by the application code. Column “Open/ Close” shows the window open/close operations triggered by the event handler. The corresponding changes to the window stack are shown in the next column. After these changes are applied, the new stack top becomes the visible window.

The first four rows in Tables 3.1/3.2 represent an event for a widget v in an activity a. If the window stack changes (rows 2–4), the callback sequences interleave lifecycle callbacks for a and the activity a' which becomes the new stack top. The implicit handlers for default events also may trigger stack changes: for example, rotating the screen destroys a and then recreates it on top of the stack (row 6). Rows 12–18 present

Table 3.3: Complete window stack changes and callback sequences (part 1).

	Stack	Event	Handler	Open/Close	Stack changes
1	$(\ldots, a)$	$[v,t]$	$[v,h]$	none	none
2	$(\ldots, a)$	$[v,t]$	$[v,h]$	$open(a')$	push a'
3	$(\ldots, a)$	$[v,t]$	$[v,h]$	$open(d)$	push d
4	$(\ldots, a)$	$[a,menu]$	implicit	$open(m)$	push m
5	$(\ldots, a)$	$[v,t]$	$[v,h]$	$open(m)$	push m
6	$(\ldots, a)$	$[v,t]$	$[v,h]$	$close(a)\,open(a')$	pop a push a'
7	$(\ldots, a)$	$[a,rotate]$	implicit	$close(a)\,open(a')$	pop a push a'
8	$(\ldots, a)$	$[a,home]$	implicit	none	none
9	$(\ldots, a)$	$[a,power]$	implicit	none	none
10	$(\ldots, a, a')$	$[v,t]$	$[v,h]$	$close(a')$	pop a'
11	$(\ldots, a, a')$	$[a',back]$	implicit	$close(a')$	pop a'
12	$(\ldots, d)$	$[v,t]$	$[v,h]$	none	none
13	$(\ldots, d)$	$[v,t]$	$[v,h]$	$open(d')$	push d'
14	$(\ldots, d)$	$[v,t]$	$[v,h]$	$open(m)$	push m
15	$(\ldots, d)$	$[v,t]$	$[v,h]$	$close(d)\,open(d')$	pop d push d'
16	$(\ldots, d)$	$[v,t]$	$[v,h]$	$close(d)\,open(m)$	pop d push m
17	$(\ldots, d)$	$[v,t]$	$[v,h]$	$close(d)$	pop d
18	$(\ldots, d)$	$[d,back]$	implicit	$close(d)$	pop d
19	$(\ldots, a, d)$	$[v,t]$	$[v,h]$	$open(a')$	push a'
20	$(\ldots, a, d)$	$[v,t]$	$[v,h]$	$close(d)\,open(a')$	pop d push a'
21	$(\ldots, a, d)$	$[v,t]$	$[v,h]$	$close(d)\,close(a)\,open(a')$	pop d pop a push a'
22	$(\ldots, a, d)$	$[d,rotate]$	implicit	$close(d)\,close(a)\,open(a')$	pop d pop a push a'
23	$(\ldots, a, d)$	$[d,home]$	implicit	none	none
24	$(\ldots, a, d)$	$[d,power]$	implicit	none	none
25	$(\ldots, a, a', d)$	$[v,t]$	$[v,h]$	$close(d)\,close(a')$	pop d pop a'
26	$(\ldots, m)$	$[v,t]$	$[v,h]$	$close(m)$	pop m
27	$(\ldots, m)$	$[m,back]$	implicit	$close(m)$	pop m
28	$(\ldots, m)$	$[v,t]$	$[v,h]$	$close(m)\,open(d)$	pop m push d
29	$(\ldots, a, m)$	$[v,t]$	$[v,h]$	$close(m)\,open(a')$	pop m push a'
30	$(\ldots, a, m)$	$[v,t]$	$[v,h]$	$close(m)\,close(a)\,open(a')$	pop m pop a push a'
31	$(\ldots, a, om)$	$[om,rotate]$	implicit	$close(om)\,close(a)\,open(a')$ $open(om')$	pop om pop a push a' push om'
32	$(\ldots, a, om)$	$[om,home]$	implicit	$close(om)$	pop om
33	$(\ldots, a, om)$	$[om,power]$	implicit	$close(om)$	pop om
34	$(\ldots, a, cm)$	$[cm,rotate]$	implicit	$close(cm)\,close(a)\,open(a')$	pop cm pop a push a'
35	$(\ldots, a, cm)$	$[cm,home]$	implicit	$close(cm)$	pop cm
36	$(\ldots, a, cm)$	$[cm,power]$	implicit	$close(cm)$	pop cm
37	$(\ldots, a, a', m)$	$[v,t]$	$[v,h]$	$close(m)\,close(a)$	pop m pop a

Table 3.4: Complete window stack changes and callback sequences (part 2).

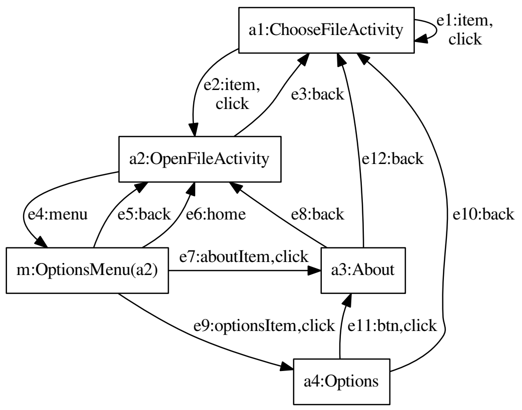
	Callback sequence
1	$[v,h]$
2	$[v,h][a,pause][a',create][a',start][a',resume][a,stop]$
3	$[v,h][d,create]$
4	$[m,create]$
5	$[v,h][m,create]$
6	$[v,h][a,pause][a',create][a',start][a',resume][a,stop][a,destroy]$
7	$[a,pause][a,stop][a,destroy][a',create][a',start][a',resume]$
8	$[a,pause][a,stop][a,restart][a,start][a,resume]$
9	$[a,pause][a,stop][a,restart][a,start][a,resume]$
10	$[v,h][a',pause][a,restart][a,start][a,resume][a',stop][a',destroy]$
11	$[a',pause][a,restart][a,start][a,resume][a',stop][a',destroy]$
12	$[v,h]$
13	$[v,h][d',create]$
14	$[v,h][m,create]$
15	$[v,h][d,destroy][d',create]$
16	$[v,h][d,destroy][m,create]$
17	$[v,h][d,destroy]$
18	$[d,destroy]$
19	$[v,h][a,pause][a',create][a',start][a',resume][a,stop]$
20	$[v,h][d,destroy][a,pause][a',create][a',start][a',resume][a,stop]$
21	$[v,h][d,destroy][a,pause][a',create][a',start][a',resume][a,stop][a,destroy]$
22	$[a,pause][d,destroy][a,stop][a,destroy][a',create][a',start][a',resume]$
23	$[a,pause][a,stop][a,restart][a,start][a,resume]$
24	$[a,pause][a,stop][a,restart][a,start][a,resume]$
25	$[v,h][d,destroy][a',pause][a,restart][a,start][a,resume][a',stop][a',destroy]$
26	$[v,h][m,destroy]$
27	$[m,destroy]$
28	$[v,h][m,destroy][d,create]$
29	$[v,h][m,destroy][a,pause][a',create][a',start][a',resume][a,stop]$
30	$[v,h][m,destroy][a,pause][a',create][a',start][a',resume][a,stop][a,destroy]$
31	$[a,pause][om,destroy][a,stop][a,destroy][a',create][a',start][a',resume][om',create]$
32	$[a,pause][om,destroy][a,stop][a,restart][a,start][a,resume]$
33	$[a,pause][om,destroy][a,stop][a,restart][a,start][a,resume]$
34	$[a,pause][cm,destroy][a,stop][a,destroy][a',create][a',start][a',resume]$
35	$[a,pause][cm,destroy][a,stop][a,restart][a,start][a,resume]$
36	$[a,pause][cm,destroy][a,stop][a,restart][a,start][a,resume]$
37	$[v,h][m,destroy][a',pause][a,restart][a,start][a,resume][a',stop][a',destroy]$

scenarios for an options menu m, owned by an activity a. The widget events [v,t] are of the form [*menu_item*,*click*] with handlers h illustrated by `onOptionsItemSelected` in the running example. The implicit $close(m)$ operation in h is explicitly represented in the table. Row 13 corresponds to lines 28–29 in the running example, and row 15 represents the effects of lines 30–32.

Window transition graph. The WTG is defined as $G = (\mathbf{Win}, E, \epsilon, \delta, \sigma)$ with nodes $w \in \mathbf{Win}$ and edges $e \in E \subseteq \mathbf{Win} \times \mathbf{Win}$. Here we use **Win** and **View** to denote sets of static abstractions of run-time windows and widgets (while previously these sets denoted the actual run-time entities). There are various ways to define such static abstractions. We use the approach from [45, 56], which creates a separate $a \in \mathbf{Win}$ for each activity class, together with appropriate $m, d \in \mathbf{Win}$ for its menus and dialogs, and abstractions $v \in \mathbf{View}$ for their widgets (i.e., defined in layout XML files), and then propagates them similarly to interprocedural points-to analysis, but with special handling of Android API calls.

Labels $\epsilon : E \rightarrow \mathbf{Event}$ indicate that the window transition represented by an edge could be triggered due to a particular event. Labels $\delta : E \rightarrow (\{push, pop\} \times \mathbf{Win})^*$ annotate an edge with a sequence of window stack operations $push(w)$ and $pop(w)$. Finally, $\sigma : E \rightarrow ((\mathbf{Win} \cup \mathbf{View}) \times \mathbf{Cback})^*$ shows the sequence of callbacks for the transition.

The meaning of an edge $e = w_1 \rightarrow w_2$ is as follows: suppose that the currently-visible window is w_1 (i.e., it is on top of the window stack). If event $\epsilon(e)$ is issued by the GUI user, the processing of this event may trigger the stack changes described by $\delta(e)$, resulting in a new stack top element w_2. During these changes, the callback sequence $\sigma(e)$ is observed.



(a) Window transition graph

e	$\delta(e)$	$\sigma(e)$	e	$\delta(e)$	$\sigma(e)$
e_1	–	1	e_7	pop m, push a_3	13
e_2	push a_2	2	e_8	pop a_3	5
e_3	pop a_2	5	e_9	pop m, pop a_2, push a_4	15
e_4	push m	9	e_{10}	pop a_4	5
e_5	pop m	16	e_{11}	pop a_4, push a_3	4
e_6	pop m	18	e_{12}	pop a_3	5

(b) Edge labels

Figure 3.2: WTG for the running example.

Example. Figure 3.2 shows the WTG for the running example. To simplify the figure, edges $w \to w$ for *rotate* and *home* events are not shown. Since edges for *power* are very similar to the ones for *home*, they are not shown either. The *back*-event edge from the starting activity a_1, which returns control back to the Android platform, is also not shown. Each e_i is labeled with its triggering event $\epsilon(e_i)$. Edge e_1 represents the case when the PDF file does not exist (line 14 in `onItemClick`) and the event handler returns without opening a new window. The table shows the associated stack changes as well as row numbers from Table 3.2 describing the callback sequences $\sigma(e)$.

Two acyclic paths reach a_3: $p = e_2, e_4, e_7$ and $p' = e_2, e_4, e_9, e_{11}$, where p produces a window stack (a_1, a_2, a_3) and p' produces (a_1, a_3). Edges e_8 and e_{12} correspond to possible next edges along p and p', respectively. Note that if e_8 is appended to p', the path is invalid: it represents a stack (a_1), but the end node of the path is a_2, which violates the property that the current window is on top of the window stack. Similarly, e_{12} cannot be appended to p. Our graph construction creates both e_8 and e_{12}, while our subsequent path traversal avoids the infeasible paths p', e_8 and p, e_{12}.

3.2 WTG Construction Algorithm

The static analysis algorithm to construct the WTG takes as input all $w \in \mathbf{Win}$, $v \in \mathbf{View}$, and, for each w, the possible widget events $[v,t]$ and their corresponding event handler callbacks $[v,h]$. This information is computed by an existing static analysis described in [45,56]. Given this input, the algorithm proceeds in three stages. In the first stage, initial edges e are constructed and annotated with trigger-event labels $\epsilon(e)$. This stage requires analysis of *open*(w) operations in event handlers, as well as modeling of default events *rotate*, *home*, *power*, and *menu*. Since *close*(w) operations are not accounted for in this stage, some of the resulting edges have incorrect target nodes. In the second stage, the initial edges are extended to include push/pop sequences $\delta(e)$ and callback sequences $\sigma(e)$. This requires analysis of self-close and owner-close operations. In the third stage, backward traversal of the graph is used to analyze the push/pop sequences along traversed paths, in order to determine the correct target nodes of edges that could not be resolved earlier.

Algorithm 3.1: ConstructInitialEdges

```
foreach $w \in$ Win do
    foreach $widget\ event\ [v,t]\ with\ callback\ [v,h]\ for\ w$ do
        if $MayOpenNone([v,h])$ then
            AddEdge($w, w, [v,t]$)
        foreach $open(w') \in Open([v,h])$ do
            AddEdge($w, w', [v,t]$)
    if $w\ is\ an\ activity\ a\ with\ options\ menu\ m$ then
        AddEdge($a, m, [a,menu]$)
    AddEdge($w, w, [w,back]$)
foreach $menu\ and\ dialog\ w \in$ Win do
    FindOwner($w$)
foreach $window\ w \in$ Win do
    if $w\ is\ an\ activity\ a$ then
        AddEdge($a, a, [a,rotate]$)
        AddEdge($a, a, [a,home]$)
        AddEdge($a, a, [a,power]$)
    if $w\ is\ an\ options\ menu\ m\ with\ owner\ a$ then
        AddEdge($m, m, [m,rotate]$)
        AddEdge($m, a, [m,home]$)
        AddEdge($m, a, [m,power]$)
    if $w\ is\ a\ context\ menu\ m\ with\ owner\ a$ then
        ...
    if $w\ is\ a\ dialog\ d\ with\ owner\ a$ then
        ...
```

3.2.1 Stage 1: Open-Window Operations and Default Events

In Stage 1, helper function AddEdge(w_1, w_2, ev) represents the addition to the WTG of an edge from window w_1 to window w_2. The edge is labeled with event *ev*: a widget event [v,t], where v is an widget in w_1, or a default event [w_1,t].

The first stage of the analysis applies Algorithm 3.1. For each window w, in addition to w's widget events [v,t] and their callbacks [v,h], the algorithm requires two additional properties. The first is a map *Open*, mapping each callback [v,h] to the set of *open*(w') operations that could be triggered by paths in the callback's execution. The second is a map *MayOpenNone* from $[v,h]$ to a boolean value: true if the callback's execution could complete without triggering any $open(w')$ (i.e., there is an execution path without window-open operations), and false otherwise. Both of these maps can be computed using an approach from [60], in which interprocedural control-flow traversal of h (and its transitive callees) is performed to find calls such as `startActivity`.

Algorithm 3.1 considers each event and its callback. If $[v,h]$ could be executed without opening a new window, an edge $w \rightarrow w$ is created. Edge e_1 in Figure 3.2 illustrates this case; the edge is created because there is a path in `onItemClick` (through line 14 in Figure 3.1) for which no windows are created. We will refer to such edges as *no-open* edges. Next, each possibly-opened window w' is considered. At line 6, an edge from w to w' is created for event $[v,t]$. Line 8 handles default event *menu* for activities. The edges created at lines 6 and 8 push a new window on top of the window stack, and will be referred to as *window-open* edges.

At line 9, initial edges for *back*-button events are created. The targets of these edges (as well as their callback sequences) will not be known until Stage 3. Next, for each menu and dialog w, its owner activity is determined by traversing backward the newly-created window-open edges, using helper function FINDOWNER.[7] Finally, default events *rotate*, *home*, and *power* are handled. This handling is consistent with the description in Table 3.1, 3.2. The cases for context menus and dialogs are not shown, but they are similar to those for options menus.

Example. Figure 3.3 shows the WTG for the running example after Stage 1 has completed. The edge numbering is the same as in the final WTG from Figure 3.2.

[7]In general, w could have multiple owners, e.g., due to subclassing of activities; the necessary algorithmic generalizations are straightforward.

Similarly to that earlier figure, certain *rotate*, *home*, and *power* edges are not shown for simplicity. Edge e_1 is created because *MayOpenNone* is true for the corresponding event handler, while e_2 shows that this handler could open a_2. The owner of m is a_2, and the *home* edge for m reflects that. The *back*-event edges have incorrect targets that will be fixed later. The *back*-event edge for a_3 is labeled as e_8, e_{12} since eventually it will lead to the creation of two separate edges e_8 and e_{12}.

After generating the edges triggered by event handling callbacks, we perform a similar analysis on the creation lifecycle callbacks of activities, in order to detect the windows opened by these methods. Based on our observations, opening/closing windows inside of lifecycle callback `onCreate` is commonly used by developers to handle exceptional run-time behavior. Consider the following example: the creation of the main activity, executed when an application is launched, triggers the associated lifecycle callback `onCreate` which attempts to initialize the network connection to the server. This operation checks if the current version of the application is obsolete in order to decide whether to an update is needed. A dialog will be displayed to the users if the network is not available. Modeling such transitions requires additional changes to Algorithm 3.1. For each edge $w_1 \rightarrow w_2$ generated by this algorithm, analysis on lifecycle callback `onCreate` of w_2 is performed to detect if it opens additional windows w_3. Then a new edge $w_1 \rightarrow w_3$ is created. The corresponding label sequence for this new edge would contain *push* w_2 followed by *push* w_3 (generated in Stage 2, as described later). If *MayOpenNone* returns false for `onCreate`, edge $w_1 \rightarrow w_2$ will be deleted indicating that this lifecycle callback will definitely trigger new windows.

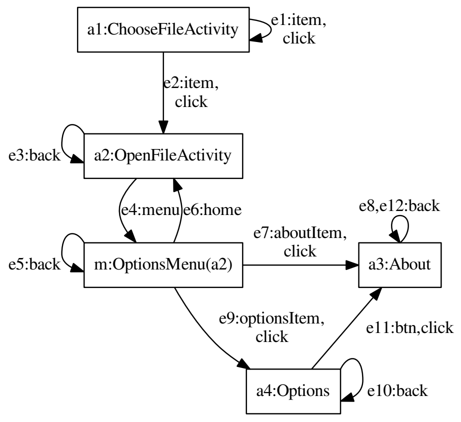


Figure 3.3: WTG after Stage 1.

3.2.2 Stage 2: Close-Window Operations

In this stage the analysis first considers each edge e for a widget event $[v,t]$ and handler $[v,h]$. Using the interprocedural control-flow reachability analysis from [60], h under calling context v is analyzed for self-close operations (e.g., calls to `finish`) and e is classified in one of three disjoint categories: must-not-self-close, may-self-close, and must-self-close. If h under context v does not contain a path reaching a self-close operation, e is in the first category. If some but not all paths reach a self-close, the second category applies. If every path reaches a self-close, the edge is must-self-close.

In a similar manner, classification is performed for owner-close operations. The analysis considers each menu and dialog w and w's owner activity a. For an edge $e = w \rightarrow \ldots$ for a widget event $[v,t]$, we can classify e as must-not-close-owner, may-close-owner, and must-close-owner.

Example. In Figure 3.3, e_7 and e_9 are must-self-close due to the implicit $close(m)$ in `onOptionsItemSelected`. Edge e_{11} is also must-self-close due to the call to `finish` at line 39 in the running example. (If, hypothetically, this call were guarded by a

Algorithm 3.2: ExpandEdgesWithLabels

```
1  foreach $w \in$ Win do
2      if $w$ is an activity $a$ then
3          foreach window-open edge $e = a \rightarrow w'$ do
4              if $e$ is may/must-self-close then
5                  ExpandEdge($e$, [pop $a$, push $w'$])
6              if $e$ is not must-self-close then
7                  ExpandEdge($e$, [push $w'$])
8          foreach no-open edge $e = a \rightarrow a$ do
9              if $e$ is may/must-self-close then
10                 ExpandEdge($e$, [pop $a$])
11             if $e$ is not must-self-close then
12                 ExpandEdge($e$, [ ])
13         if exists $e = a \rightarrow m$ for default event [$w$,menu] then
14             ExpandEdge($e$, [push $m$])
15     if $w$ is a menu $m$ with owner $a$ then
16         foreach window-open edge $e = m \rightarrow w'$ do
17             if $e$ is may/must-owner-close then
18                 ExpandEdge($e$, [pop $m$, pop $a$, push $w'$])
19             if $e$ is not must-owner-close then
20                 ExpandEdge($e$, [pop $m$, push $w'$])
21         foreach no-open edge $e = m \rightarrow m$ do
22             if $e$ is may/must-owner-close then
23                 ExpandEdge($e$, [pop $m$, pop $a$])
24             if $e$ is not must-owner-close then
25                 ExpandEdge($e$, [pop $m$])
26     if $w$ is a dialog $d$ with owner $a$ then
27         ...
28 foreach edge $w \rightarrow w$ for default event [$w$,back] do
29     ExpandEdge($e$, [pop $w$])
```

conditional, the classification would have been may-self-close.) The other two widget event edges e_1 and e_2 are must-not-self-close. For owner-close operations, e_7 is must-not-close-owner, while e_9 is must-close-owner, since under widget context `optionsItem` the handler definitely closes the owner activity a_2 (line 32 in the running example).

This classification is used to create push/pop labels $\delta(e)$ for the analyzed edges. For example, e_9 opens a_4 while definitely closing m and its owner a_2; thus, $\delta(e_9) =$ *pop* m, *pop* a_2, *push* a_4. Algorithm 3.2 provides some details on this process. One important observations is that a single edge created by Stage 1 may be expanded into several edges, with different $\delta(e)$ labels. For example, if (hypothetically) e_{11} were may-self-close, it would expand to two edges from a_4 to a_3, one labeled with *push* a_3 (line 7 in the algorithm) and the other with *pop* a_4, *push* a_3 (line 5 in the algorithm). Helper function $\textsc{ExpandEdge}(e, d)$ takes an edge e created by Stage 1 and constructs an "expanded" version of it with $\delta(e) = d$. After all expansions done in Stage 2, the edges from Stage 1 are discarded.

As discussed earlier, Stage 1 considers the open-window effects of `onCreate` life-cycle callbacks for activities. Similarly, Stage 2 analyzes the close-window effects of these callbacks. To capture such close operations for edges triggering new activities, two cases are considered. First, if the opened activity must be closed by its `onCreate` (i.e., every path contains a self-close operation), the corresponding *pop* will be included in the edge's label. Otherwise, if the target activity may be closed (i.e., some but not all paths contain self-close operations), two edges are generated: one with and one without a *pop* label. For example, if transition $w_1 \rightarrow w_2$ must close target activity w_2, one edge with label *push* w_2 *pop* w_2 will represent this behavior. If w_2 may be closed during this transition, two edges will be created: one with label *push* w_2 and one with label *push* w_2 *pop* w_2. Recall that in some cases a transition $w_1 \rightarrow w_2$ may be created by Stage 1 when `onCreate` of some intermediate activity a opens w_2 (and a itself is opened by w_1). In this case the analysis will create either a single edge with label *push a push* w_2 *pop* w_2, or two edges with labels *push a push* w_2 and *push a push* w_2 *pop* w_2.

The handling of dialogs is similar to that of menus, but with the additional possibility that a self-close operation is not executed. The handling of *rotate*, *home*, and *power* events is consistent with the push/pop sequences listed in Table 3.1, 3.2, and is not shown in Algorithm 3.2. After the algorithm completes, all edges have labels $\delta(e)$. The labels created for the running example are shown in Figure 3.2b. At this point, there is still a single *back*-event edge for a_3 (labeled with *pop* a_3); Stage 3 creates two separate edges from it.

Certain edges have incorrect targets and have to be processed by Stage 3. These edges do not open new windows, but close existing ones: namely, (1) edges for *back* events, and (2) no-open edges that contain close operations. In both cases, the top of the stack after executing the edge is some (yet) unknown previously-opened window. The rest of the edges have correct target nodes and their callback sequences $\sigma(e)$ can be determined at this time, using the Android semantic specification illustrated by Table 3.1, 3.2. For edges $e_1, e_2, e_4, e_6, e_7, e_9, e_{11}$ from Figure 3.3, the callback sequences computed by Stage 2 are listed in Figure 3.2b. The rest of the edges in Figure 3.3 have incorrect target nodes, and since $\sigma(e)$ depends on the target of e, their callback sequences cannot yet be determined.

3.2.3 Stage 3: Backward Analysis of the Window Stack

Edges with incorrect targets require further processing. They are of the form $e = w \rightarrow w$, with labels $\delta(e)$ containing no *push* but at least one *pop*. To identify the correct target of e, Stage 3 performs a backward traversal from w, using correct edges finalized in Stage 2, to examine all paths ending at w. This traversal is parameterized by a value k, which defines the largest number of edges along any path being considered.[8] For each such path $e_1, e_2, \ldots, e_n$, where $n \leq k$ and the target node of e_n is w, we need to consider the sequence of push/pop operations $\delta(e_1), \delta(e_2), \ldots, \delta(e_n), \delta(e)$ and to decide (1) whether this sequence represents valid run-time behavior, and (2) what could be the top of the window stack after the sequence is executed.

Example. Suppose that $k = 2$ and we consider $e_5 = m \to m$ in Figure 3.3, labeled with *pop* m. Two paths ending at m need to be examined: e_2, e_4 and e_6, e_4. The edge labels for the first path (including e_5's label) are *push* a_2, *push* m, *pop* m. This is a feasible sequence whose execution is guaranteed to leave a_2 as the top of the stack. Thus, e_5 should have a_2 as a target, and the analysis creates this corrected edge. For the second path, the edge labels (including e_5) are *pop* m, *push* m, *pop* m. Although this is a feasible sequence, it does not provide enough information to decide what would be the top of the stack after executing these operations, and the analysis does not create any edges due to this path.

As another example, consider edge $e_{10} = a_4 \to a_4$. For $k = 4$, the relevant path is e_0, e_2, e_4, e_9. Here e_0 is an implicit edge entering a_1, labeled with *push* a_1; this edge represents the triggering of the start activity a_1 by the Android platform. The sequence for $e_0, e_2, e_4, e_9, e_{10}$ is *push* a_1, *push* a_2, *push* m, *pop* m, *pop* a_2, *push* a_4, *pop* a_4. This sequence leaves a_1 as the top of the stack. Thus, e_{10} should be redirected to a_1 (as shown in the graph in Figure 3.2).

As a final example, consider *back*-event edge $a_3 \to a_3$. Path e_2, e_4, e_7, with this edge appended, has the sequence *push* a_2, *push* m, *pop* m, *push* a_3, *pop* a_3. Thus,

[8]An alternative would be to traverse all acyclic paths, without a length limit.

this *back*-event edge should have a_2 as target. In the final graph from Figure 3.2, e_8 is this redirected edge. Another relevant path is $e_0, e_2, e_4, e_9, e_{11}$; the sequence along the path, appended with the *back*-event edge, is *push* a_1, *push* a_2, *push* m, *pop* m, *pop* a_2, *push* a_4, *pop* a_4, *push* a_3, *pop* a_3, which leaves a_1 as the top of the stack. In this case an edge from a_3 to a_1 needs to be introduced (e_{12} from Figure 3.2).

Stage 3 analyzes an edge $e = w \rightarrow w$ as follows. A stack containing *push* and *pop* operations is maintained. The stack is initialized with the reverse of $\delta(e)$; for all examples from above, this is an operation *pop* w. Backward traversal from w is performed, limiting path length to at most k edges. When an edge e_i is encountered during the traversal, the reverse of its $\delta(e_i)$ sequence is used to update the stack. If *pop* w' is seen, it is just added on top of the stack. If *push* w' is encountered and the stack is not empty, the top of the stack must be *pop* w' (otherwise the path is infeasible and is ignored) and *pop* w' is removed from the stack. If *push* w' is observed when the stack is empty, the traversal stops and w' is identifies as a possible target, leading to a new edge $w \rightarrow w'$. After these edges are changed with the correct targets, their callback sequences $\sigma(e)$ can be generated in a similar way described previously.
Example. Consider edge $e_{10} = a_4 \rightarrow a_4$. Starting from a stack containing *pop* a_4, edges e_9, e_4, e_2, e_0 are visited to produce the following sequence: *push* a_4, *pop* a_2, *pop* m, *push* m, *push* a_2, *push* a_1. Operations *push* a_4 and *push* a_2 empty the stack. Since *push* a_1 occurs for an empty stack, edge e_{10} becomes $a_4 \rightarrow a_1$.

3.2.4 Limitations

The algorithm and its implementation have several limitations. As discussed earlier, control flow due to multiple threads or across multiple applications is not modeled. The modeling of GUI widgets and event handlers [45] captures many commonly-used Android widgets, but is not fully comprehensive. Furthermore, custom window/widget systems cannot be handled. Asynchronous transitions (e.g., due to timers and sensor events) are not represented in the WTG. The interprocedural intent analysis used to resolve $open(w)$ calls [60] considers only explicit intents, as they are designed for use inside the same application [23]. More general intent analyses (e.g., [35,36,47]) could be used instead. Our analysis also does not model the different launch modes for activities [2]. Due to these limitations, some window transitions are missing: for example, for the 20 apps used in our evaluation, on average 13% of the WTG nodes have no incoming edges. While most of these limitations are orthogonal to the contributions of this chapter, they emphasize the need to advance the state of the art in static analysis for Android, and in particular the comprehensive modeling of Android-specific control flow and data flow.

3.2.5 Path Validity

The analysis outlined in the previous sections does not ensure that each path represents a feasible run-time execution. Consider again the final WTG (after Stage 3) shown in Figure 3.2. Paths $p = e_0, e_2, e_4, e_7$ and $p' = e_0, e_2, e_4, e_9, e_{11}$ both reach node a_3. However, p cannot be extended with edge e_{12} because the corresponding edge labels would be *push* a_1, *push* a_2, *push* m, *pop* m, *push* a_3, *pop* a_3. This leaves a_2 as the top of the window stack, while the target node of e_{12} is a_1. Similarly, if p' were extended with e_8, the top of the stack would be a_1 while the target of e_8 is a_2.

The WTG can be augmented with a path validity check, which "simulates" the window stack along a given path of interest, and decides whether the path is valid.

This is similar in spirit to classical interprocedural analyses, where the sequence of calls and returns along a path is used to simulate the call stack, in order to decide path validity [48]. A WTG edge may correspond to several push/pop operations, but the validity of these operations is still based on the same style of push/pop matching as in traditional analyses. As discussed in the next section, one use of this validity check is during test generation, to avoid the creation of unexecutable test cases. Path validity checks may also be needed for static checking of correctness properties, in order to avoid analyzing infeasible paths that lead to false positives.

3.3 Test Generation

One possible application of the WTG is for model-based test generation (e.g., [8, 24, 52, 58, 59]). To illustrate this use of the WTG, we developed a prototype test generation tool. The tool traverses certain WTG paths and for each path creates a test case implemented with the Robotium testing framework [43]. For a path $p = e_1, e_2, \ldots$, the event label $\epsilon(e_i)$ is translated to corresponding Robotium API calls to trigger the event. Some events may require additional input from the tester—e.g., to decide which item in a list to click. Since the static analysis solution is conservative, it is possible that event $\epsilon(e_i)$ may not be feasible at run time, or even if it is feasible, the target window of e_i after the run-time event is not as expected. Each test case includes run-time checks to detect such scenarios and report the test case as infeasible.

One can consider various test generation schemes (e.g., leak testing in [58] considers neutral-effect cycles in a manually-constructed model). In our proof-of-concept tool, we use a simple path-based approach. Starting from the implicit edge e_0 showing the invocation of the start activity, we append m distinct edges to create a path

Table 3.5: WTG construction algorithm: number of nodes/edges across stages.

Application	SLOC	Nodes	Edges				
			Stage 1	Stage 2	$\Delta_{1,2}$	Stage 3	$\Delta_{2,3}$
`APV`	3832	14	77	101	24	105	58
`Astrid`	24487	93	594	740	146	838	236
`BarcodeScanner`	6549	20	90	121	31	128	65
`Beem`	12962	24	99	125	26	132	65
`ConnectBot`	32638	37	185	233	48	237	112
`FBReader`	45510	45	286	17774	17488	41942	26326
`K9`	52240	55	258	411	153	516	221
`KeePassDroid`	27457	41	272	468	196	643	389
`Mileage`	9881	75	409	562	153	676	268
`MyTracks`	23389	61	212	314	102	391	197
`NotePad`	4986	22	122	191	69	213	110
`NPR`	12118	32	344	502	158	590	106
`OpenManager`	2562	18	95	116	21	116	64
`OpenSudoku`	6079	35	173	232	59	237	125
`SipDroid`	24533	31	176	305	129	406	331
`SuperGenPass`	2119	9	49	63	14	64	39
`TippyTipper`	1739	10	54	63	9	65	16
`VLC`	10670	26	117	130	13	131	45
`VuDroid`	2380	7	30	44	14	47	23
`XBMC`	23295	67	1080	3819	2739	4279	722

$p = e_0, e_1, \ldots, e_m$. A naive approach is to simply explore all such paths. A more precise approach is to apply the validity check from Section 3.2.5 each time the path is extended with a new edge. The next section shows that this validity check, which is based on our proposed tracking the push/pop sequences, can reduce substantially the number of test cases being generated.

3.4 Experimental Evaluation

The WTG was constructed for the 20 open-source applications used in the previous chapter. The first goal of the evaluation is to characterize the effects of different stages

Table 3.6: WTG construction algorithm: search depth and analysis cost.

Application	k				Time
	$k=1$	$k=2$	$k=3$	$k=4$	(sec)
`APV`	75	95	104	105	5
`Astrid`	675	836	838	838	18
`BarcodeScanner`	98	118	128	128	6
`Beem`	100	118	132	132	6
`ConnectBot`	182	211	234	237	8
`FBReader`	29473	39820	41941	41942	2086
`K9`	433	486	509	516	25
`KeePassDroid`	399	598	640	643	9
`Mileage`	485	636	676	676	7
`MyTracks`	294	363	391	391	7
`NotePad`	162	195	213	213	6
`NPR`	525	590	590	590	6
`OpenManager`	84	113	116	116	5
`OpenSudoku`	180	208	237	237	6
`SipDroid`	226	364	396	406	12
`SuperGenPass`	45	58	64	64	5
`TippyTipper`	56	61	65	65	5
`VLC`	112	131	131	131	6
`VuDroid`	34	41	47	47	4
`XBMC`	3690	4241	4278	4279	16

of the algorithm, as well as its overall cost. The second goal is to evaluate precision, relative to a manually-constructed model. The third goal is to evaluate precision for the test generation from Section 3.3. The implementation is available as part of our public analysis toolkit [15].

3.4.1 Algorithm for Building the WTG

Tables 3.5/ 3.6 provide measurements of the number of WTG nodes and edges. Column "Stage 1" shows the number of edges before considering any close-window operations (Algorithm 3.1). After Stage 2, the edges are expanded with push/pop sequences, based on analysis of close-window effects. Column $\Delta_{1,2}$ shows the increase due to this expansion. One can observe that an edge from Stage 1 can often have several possible push/pop sequences. This indicates that an event handler may exhibit a variety of behaviors. Our analysis discovers such variations and represents them with separate edges (Algorithm 3.2). We are not aware of any existing work that performs such detailed analysis of Android event handlers.

The large number of edges for `FBReader` and `XBMC` is caused by a known limitation of our prior analyses [45, 60]: both analyses use a context-insensitive call graph based on class hierarchy analysis. For example, in `FBReader`, two utility methods are responsible for over 96% of the WTG edges. Both methods take parameters of `Runnable` type which is implemented by 130 classes. Class hierarchy resolution for calls on these parameters is highly imprecise. The next chapter discusses this program in more detail.

Recall that some of the Stage 2 edges have incorrect target nodes. Column "Stage 3" shows the number of edges after the correct targets have been determined. This is achieved with backward path analysis, based on a parameter k for path length; the column of Table 3.6 contains measurements for $k=4$. Column $\Delta_{2,3}$ in Table 3.5 shows the size of the difference (number of edges removed and added) between the edge sets from Stage 2 and Stage 3. The backward path traversal, combined with tracking of feasible push/pop sequences along the path (Section 3.2.3), results in significant changes to the graph. The four columns of Table 3.6 show the effects of increasing the path length limit k. In general, newly-created edges require backward traversals of non-trivial length. Thus, one cannot consider just the edges entering a node w to determine the targets of Stage 3 edges $w \rightarrow \ldots$; rather, paths of length k reaching w

Table 3.7: Feasibility of WTG edges.

Application	WTG	Manual	Infeasible
APV	105	105	0
BarcodeScanner	128	106	22
OpenManager	116	109	7
SuperGenPass	64	64	0
TippyTipper	65	65	0
VuDroid	47	45	2

must be examined. To the best of our knowledge, ours is the first approach to perform such static modeling of possible transitions in Android GUIs. For most programs, the graph stabilizes at $k=4$; for the rest, slightly larger values of k (not shown here) are needed.

The last column shows the running time of the analysis in seconds. This measurement includes the time for the event handler analysis from [60], which is invoked on-demand inside our analysis. Overall, the running times are suitable for practical use, even though we have not made any significant effort to optimize the implementation. However, as indicated by the results for `FBReader`, scalability limitations could be encountered for large WTGs.

3.4.2 Manual Examination of WTGs

For in-depth evaluation of analysis precision, we examined the WTG ($k = 4$) for `APV`, `BarcodeScanner`, `OpenManager`, `SuperGenPass`, `TippyTipper`, and `VuDroid`. These applications had the smallest numbers of WTG nodes, and thus could be examined manually with reasonable effort.

Column "WTG" in Table 3.7 replicates the Stage 3 measurements from Tables 3.5/3.6. Column "Manual" shows the number of WTG edges that were manually confirmed to be feasible using run-time test cases. The last column contains the number of infeasible WTG edges. The infeasibility was asserted by examining the source code. In general, the number of infeasible edges is small (around 6% across the six applications). We determined the root causes of all infeasible edges. In all cases, the infeasibility was due to deficiencies in the earlier work on window/widget modeling [45, 56] and event handler analysis [60]. If these existing static analyses were to be improved, the WTG would achieve perfect precision. These results highlights the need for continued advances in static analysis of GUI structure and behavior for Android applications. Still, the small number of infeasible edges is a positive indicator that highly-precise static GUI models can be constructed automatically.

3.4.3 Test Generation

Recall that our prototype test generator considers paths $p = e_0, e_1, \ldots, e_m$ (all e_i are distinct) and generates test cases from them. Here e_0 represents the invocation of the start activity by the Android platform. The numbers of paths for $m=2$ and $m=3$ are shown in Table 3.8. Columns "All" contain the number of all paths, while columns Δ show the reduction (in percent) when the path validity check from Section 3.2.5 is applied. For `FBReader` the number of paths with $m = 3$ was too large enumerate in reasonable time.

For several applications the path validity check reduces the number of test cases. For example, for $m=3$ (i.e., test cases containing three GUI events), 10 applications

Table 3.8: Number of paths for test generation.

Application	$m=2$		$m=3$	
	All	Δ (%)	All	Δ (%)
`APV`	116	24.1	1416	37.7
`Astrid`	232	62.1	1822	75.3
`BarcodeScanner`	526	1.9	7675	4.6
`Beem`	138	26.1	929	38.6
`ConnectBot`	287	26.1	3384	40.3
`FBReader`	33404638	84.9	N/A	N/A
`K9`	12393	19.8	443647	27.0
`KeePassDroid`	20	0.0	48	0.0
`Mileage`	16	0.0	45	0.0
`MyTracks`	1331	9.4	35212	20.5
`NotePad`	217	17.5	2625	26.0
`NPR`	4171	21.0	251251	30.5
`OpenManager`	392	0.8	5803	1.5
`OpenSudoku`	111	23.4	980	33.3
`SipDroid`	905	32.9	13604	51.6
`SuperGenPass`	195	0.0	2110	0.0
`TippyTipper`	341	0.0	5405	0.0
`VLC`	42	0.0	131	0.0
`VuDroid`	52	0.0	276	0.0
`XBMC`	5728	62.3	1330605	71.0

show reductions of 26% or more. Such reductions indicate that statically we can eliminate significant numbers of infeasible test cases.

Of course, even if a path satisfies the static validity condition, it could still result in an infeasible test case. As indicated earlier, due to deficiencies in prior static analyses, some WTG edges (and thus paths) may be infeasible. To understand better this infeasibility, for the six applications studied in Section 3.4.2 we generated test cases from the statically-feasible paths for $m = 2$. Although the sequences of events (implemented through Robotium [43] API calls) are generated automatically, some

Table 3.9: Run-time feasibility of generated test cases.

Application	Static	Feasible
`APV`	88	88
`BarcodeScanner`	516	88
`OpenManager`	389	364
`SuperGenPass`	195	195
`TippyTipper`	341	341
`VuDroid`	52	52

test cases still require manual effort: for example, for `BarcodeScanner`, we need to manually set up a variety of actual barcode images to drive the different test cases. Due to this manual effort, we did not consider larger values of m.

The number of test cases (with path validity) is shown in column “Static” in Table 3.9. We set up and executed all 1581 test cases indicated in this column. The next column “Feasible” shows the number of these test cases that were feasible at run time—that is, they could match the event sequence and target windows of the static path. In `BarcodeScanner`, the event handler analysis from [60] leads to infeasible edges that make most of the test cases infeasible. As described in [60], the application processes eleven types of barcodes, and the GUI behavior (subset of visible widgets and subset of handler effects) differs based on the barcode type. This variability cannot easily be modeled statically. In `OpenManager`, the 6.5% of infeasible test cases are due to inter-application interactions. When the main activity is invoked by another application (rather than by the user), that activity computes information about a file, returns it to the invoking application, and closes itself. Our analysis does not model the interactions between multiple applications and does not recognize that the activity-close operation happens under these conditions. Overall, with the exception of one application, the vast majority of statically-generated test cases are feasible at run time.

Summary. Columns $\Delta_{1,2}$ and $\Delta_{2,3}$ of Table 3.5, 3.6 indicate that event handlers can have complex behaviors and their transitions depend on non-trivial sequences of preceding events. Our analysis is the first to model these features, leading to improved static GUI models and test case generation. For six applications, manual comparison with run-time behavior indicates that the analysis achieves good precision.

CHAPTER 4: Modeling of Asynchronous Control Flow

The WTG described in the previous chapter represents only control flow caused by code executed by the UI thread of the application. In Android, other theads may asynchronously trigger GUI changes by posting code to be executed by the UI thread. This chapter describes an extension of the WTG definition and construction to capture common cases of such behavior.

4.1 Window Transitions Triggered by Other Threads

Chapter 3 proposed the window transition graph (WTG) to describe the GUI behavior of Android applications. Recall that the WTG is defined as $G = (\mathbf{Win}, E, \epsilon, \delta, \sigma)$ with nodes $w \in \mathbf{Win}$ representing windows and edges $e \in E \subseteq \mathbf{Win} \times \mathbf{Win}$ representing window transitions. An edge e is labeled with several labels: $\epsilon(e)$ shows that the transition could be triggered due to a particular event, $\delta(e)$ annotates e with a sequence of window stack operations $push(w)$ and $pop(w)$, and $\sigma(e)$ is the sequence of callbacks that occur during the transition. For an edge $e = w_1 \rightarrow w_2$, if the top of the window stack is w_1 and event $\epsilon(e)$ is issued by the GUI user, the handling of this event may trigger the stack changes denoted by $\delta(e)$, leading to a new stack top element w_2. During these changes, the callbacks in $\sigma(e)$ are observed.

Events $\epsilon(e)$ are either widget events (e.g., clicking on a button) or default events $(back, rotate, home, power, menu)$. The sequence of such events is generated by the GUI user and is processed by the UI interface thread, which is the main thread of

```
class OpenFileActivity
   extends Activity {
 private String findText = null;
 private MenuItem findTextMenuItem = ...;
 public boolean onOptionsItemSelected(MenuItem menuItem) {
  if (menuItem == this.findTextMenuItem) {
   final Dialog dialog = new Dialog(this);
   Button goButton = new Button(this);
   goButton.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
     OpenFileActivity.this.findText = ...;
     OpenFileActivity.this.find(true);
     dialog.dismiss();
    }
   });
   dialog.setContentView(contents); // 'contents' contains goButton
   dialog.show();
  }
 }
 void find(boolean forward) {
  Finder finder = new Finder(this, forward);
  Thread finderThread = new Thread(finder);
  finderThread.start();
 }
 static class Finder implements Runnable,
  DialogInterface.OnCancelListener, DialogInterface.OnClickListener {
  private OpenFileActivity parent = null;
  private boolean forward;
  public Finder(OpenFileActivity parent, boolean forward) {
   this.parent = parent;
   this.forward = forward;
  }
  public void run() {
   this.parent.runOnUiThread(new Runnable() {
    public void run() {
     AlertDialog dialog = ...;
     dialog.show();
    }
   });
  }
 }
}
```

Figure 4.1: Example derived from the APV PDF reader [5]

the application. However, in addition to user-event-driven window transitions, the UI thread may perform transitions due to other threads in the application. Such threads can generate separate sequences of events and window transitions that are interleaved with the ones generated by the GUI user. The WTG representation described earlier is not designed to capture such events and transitions.

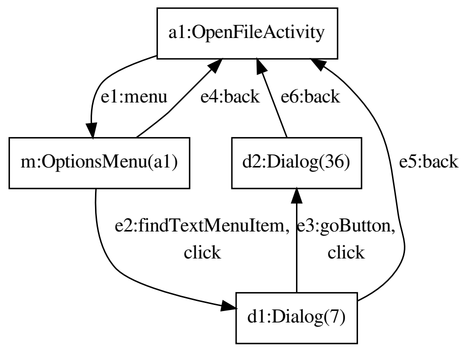
The work presented in Chapter 3 does consider two API calls allowing other threads to post events on the UI thread: `Activity.runOnUiThread` and `View.post`. However, as explained later, this handling does not represent faithfully the actual run-time execution. Furthermore, not all uses of these calls are considered, and several other similar APIs are not handled at all. The work presented in this chapter (1) generalizes the APIs that are considered by the analysis, (2) uses a different representation to integrate them in the WTG, and (3) uses a different analysis to model their effects.

We illustrate these issues with two examples derived from APV [5]. Figure 4.1 contains the first example. This example shows the use case of text search in a PDF file. When options menu item `findTextMenuItem` is selected, a dialog to enter the searched text is displayed (line 17). After the user enters the text and clicks `goButton`, this dialog is closed (line 13). Meanwhile, a thread is started searching for the text (line 23); the progress of this thread is displayed by another dialog (openend at line 37). The search functionality is offloaded to a background thread for better responsiveness, because the UI thread may be blocked while a large PDF file is processed. Because Android does not allow non-UI threads to access GUI widgets, developers use API calls such as `Activity.runOnUiThread` (line 34) to post a `Runnable` task into the event queue of the UI thread, in order to update the progress dialog. This is a typical example of how another thread can affect the execution of the UI thread and trigger window transitions.

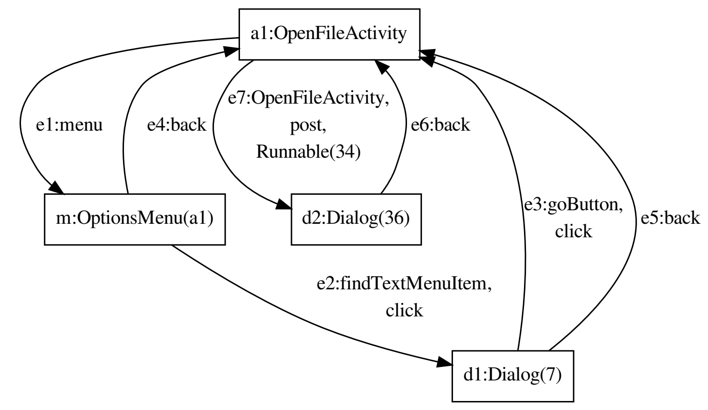
The WTG approach described in the previous chapter uses Algorithm 2.1 to detect opened or closed windows. This algorithm analyzes the interprocedural control-flow graph (ICFG) starting from event handler callbacks and lifecycle callbacks. In this example, the ICFG starting from event handler `onClick` will be constructed and analyzed. The version of the algorithm described in the previous chapter uses a specialized form of the ICFG. This form resolves calls to trigger new threads and considers the control flow due to the execution of these threads. Going back to the example of analyzing `onClick`, the call to `start` at line 23 will be considered to be an invocation of method `run` defined at lines 33–40. To identify the called methods for such invocations, a backward traversal along the chain of assignments is used to identify the `Runnable` instance that flows as a parameter to the constructor of `Thread` (line 22). This traversal is performed in the flow graph defined in prior work [45]. In this example, the thread is coupled with the `Finder` instance created at line 21. This information is used to resolve the call to `start` at line 23: the `Thread` instance that flows to the call to `start` is determined through backward traversal of the flow graph (this instance is created at line 22) and the associated runnable object is examined. As a result, in the ICFG, the call to `start` is considered to invoke `Finder.run`.

Similarly, the call to `runOnUiThread` (line 34) will be considered to be an invocation of method `run` defined at lines 35–38. This resolution is done by determining the `Runnable` instances that can flow as parameters of the call. The corresponding `run` methods will be considered as callees at this call site.[9] In the example, the parameter of `runOnUiThread` is determined to be the anonymous class instance created at line 34, and the ICFG will consider the call to `runOnUiThread` to be an invocation of method `run` defined at lines 35–38. Thus, the ICFG will include the call to `show` at line 37. As a result, the WTG will contain a transition from the window for the dialog

[9]Due to unsoundness in the modeling of the complete Android framework, it is possible that no objects are reached during the backward flow graph traversal from a `Runnable` parameter. In this case, class hierarchy analysis is used to determine the potentially invoked `run` methods.



(a) Current WTG



(b) New WTG

Figure 4.2: WTG comparison for Example 4.1

created at line 7 to the progress dialog created at line 36. This edge is shows as e_3 in Figure 4.2(a). While this approach considers the code executed by the UI thread due to other threads (i.e., the body of `run` at lines 36–37), it does not represent precisely the run-time behavior, since it implies that the body of `run` is executed as part of processing a click event.

Another example, presented in Figure 4.3, describes a similar case. In this example, `pagesView` is a view associated with activity `OpenFileActivity`. When the Android framework invokes callback `onDraw` on this view (when the view is drawn), a `PDFPagesProvider` is used to prepare the PDF file (method `setVisibleTiles`) by

```java
1  public class PDFPagesProvider extends PagesProvider {
2    private PagesView pagesView = ...;
3    private RendererWorker rendererWorker = new RendererWorker(this,...);
4    public PDFPagesProvider(PagesView view,...) {
5      this.pagesView = view;
6    }
7    public void setVisibleTiles(Collection<Tile> tiles) {
8      List<Tile> newtiles = ...;
9      this.rendererWorker.setTiles(newtiles,...);
10   }
11   public PagesView getPagesView() {
12     return this.pagesView;
13   }
14   private static class RendererWorker implements Runnable {
15     private PDFPagesProvider pdfPagesProvider = ...;
16     RendererWorker(PDFPagesProvider provider,...) {
17       this.pdfPagesProvider = provider;
18     }
19     void setTiles(Collection<Tile> tiles,...) {
20       Thread t = new Thread(this);
21       t.start();
22     }
23     public void run() {
24       while(true) {
25         try {
26           Map<Tile,Bitmap> renderedTiles = this.pdfPagesProvider.renderTiles(tiles);
27         } catch (RenderingException e) {
28           PagesView pagesView = this.pdfPagesProvider.getPagesView();
29           pagesView.post(new Runnable() {
30             public void run() {
31               AlertDialog errorMessageDialog = ...;
32               errorMessageDialog.show();
33             }
34           });
35         }
36       }
37     }
38   }
39 }
40 public class PagesView extends View {
41   private PagesProvider pagesProvider = new PDFPagesProvider(this,...);
42   public void onDraw(Canvas canvas) {
43     LinkedList<Tile> visibleTiles = ...;
44     this.pagesProvider.setVisibleTiles(visibleTiles);
45   }
46 }
```

Figure 4.3: Another example derived from APV

offloading the rendering work into a background thread `RendererWorker` at line 21. The call to `renderTiles` (line 26) renders all PDF pages unless a `RenderingException` is thrown. This error happens when there is insufficient memory to load the PDF file. In such a situation a `Runnable` instance is posted on the UI thread, in order to create an `errorMessageDialog` displaying an error message to the user. This process is done through the invocation of standard API `View.post` at line 29. In this example, the WTG described in the previous chapter does not contain an edge to represent the opening of the error message dialog: since `onDraw` is not a callback representing a widget event, it will not be analyzed and the call chain to the call of `show` at line 32 (reached from `onDraw` by resolving the calls to `Thread.start` and `View.post`) will not be examined.

4.1.1 Representation of Events and Transitions Triggered by Other Threads

The examples from the previous section exemplify a general pattern: after being offloaded into a background thread, a long running task has to communicate with the UI thread in order to access and modify GUI state. In this chapter we propose WTG generalizations and analysis to represent some common cases of such interactions. In particular, we focus on the following three standard Android API calls:

- `Activity.runOnUiThread(Runnable)`: This API enqueues the runnable task on the event queue of the UI thread. The posted task will be executed in the future when the UI thread is free.

- `View.post(Runnable)`: Similarly to `runOnUiThread`, this API call adds a runnable action to the event queue. The task will be run on the UI thread.

- `View.postDelayed(Runnable,long)`: Calling this method will add the runnable task into the event queue. The task will be run on the UI thread once the specified amount of time elapses.

We aim to model the behavior of these APIs (together with the `Handler` APIs described later) in a more comprehensive and precise manner compared to the work presented in the previous chapter. This extension could be used as the starting point for more generalized control-flow analyses of Android asynchronous operations, beyond these particular API calls.

To represent the run-time behavior of such asynchronous operations, we consider and analyze the `run` methods of runnable tasks used as parameter at calls to `runOnUiThread`, `post`, and `postDelayed`. Each such `run` method can be thought of as a special kind of event handler for an artificial "post" event. The post event is triggered by another thread (not the UI thread), concurrently with the "regular" widget/default events triggered by the GUI user. The representation of these artificial events and their `run` handlers is detailed next.

Runnable tasks. Classes implementing interface `java.lang.Runnable` define runnable objects that encapsulate tasks to be run in various threads of the application. For the two examples discussed earlier, classes `Finder` and `RendererWorker` represent tasks to be executed in non-UI threads, while the two anonymous inner classes implementing `Runnable` define tasks to be executed in the UI thread. Let **Runnable** be the set of all static abstractions of runnable objects. As usual, in our analysis each such abstraction corresponds to a `new` expression instantiating a `Runnable` class.

Post events. We define a special type of "post" event to represent the posting of a runnable task to be executed by the UI thread. A post event will be represented as $e = [w,\mathit{post},r]$ where $w \in \mathbf{Win}$ indicates the window through which the event is posted and $r \in \mathbf{Runnable}$ is the posted runnable task. For example, in Figure 4.1, the call to `runOnUiThread` at line 34 triggers event $e = [\mathit{OpenFileActivity},\mathit{post},\mathit{Runnable}(34)]$.

Here $\mathit{Runnable}(34)$ is a static abstraction of the object created at that line. For Figure 4.3, line 29 corresponds to $e = [\mathit{OpenFileActivity}, \mathit{post}, \mathit{Runnable}(29)]$. In this case `pagesView` is part of the GUI hierarchy of `OpenFileActivity`, and this activity is considered to be the window through which the event is posted.

Source window. Post events may trigger window transitions that are represented by edges in the WTG. The source node of a transition for an event $e = [w, \mathit{post}, r]$ will be considered to be w. In our observations, the typical scenario is the following: posting of a runnable task on the UI thread is done through the window that is on top of the window stack, and by the time the task is actually executed this window is still on top of the stack. Thus, in our WTG representation and analysis, we assume that the visible window at the time the task is posted is the same as the visible window at the time the task is executed. Future work may consider generalizations of this approach, where the two windows may potentially be different.

Figure 4.2b shows the new WTG for Example 4.1 after introducing post events. The target of edge e_3 in the new WTG is changed to `OpenFileActivity` because this edge represents an event handler that closes the current dialog (line 13 in Figure 4.1) and starts a new `Finder` thread. A new edge e_7 is added in the WTG to represent the effects of the post event triggered at line 34. Note that the two edges are related: e_7 can be observed only after e_3 has occurred. In the current WTG representation we do not explicitly represent this constraint. However, such ordering constraints can be easily produced by our static analysis (described shortly) and can be utilized by client analyses—for example, to enumerate valid WTG paths for automated test generation or static checking.

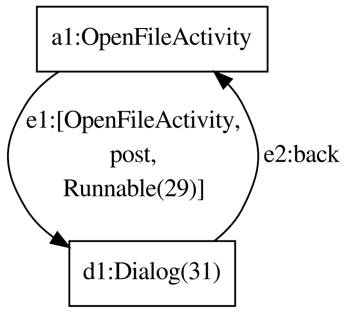


Figure 4.4: WTG for the example in Figure 4.3

In the example from Figure 4.3, the post event is not analyzed by the approach from the previous chapter, since `onDraw` does not correspond to a widget event. However, with the new version of the WTG, we will explicitly analyze the call to `post` and will determine that it corresponds to $e = [\mathit{OpenFileActivity}, \mathit{post}, \mathit{Runnable}(29)]$. The analysis of the `run` method for the runnable task will determine that an error message dialog may be opened. The resulting WTG is shown in Figure 4.4.

4.2 Algorithmic Extensions

The WTG construction algorithm described in Section 3.2 is composed of three stages. In the first stage, edges are constructed for open-window and default events. Since close-window operations are not analyzed at this stage, some of the generated edges may be incorrect. Owner relationships also computed by this first stage. The second stage extends the initial edges to include self-close and owner-close operations. Additional information about push/pop sequences $\delta(e)$ and callback sequences $\sigma(e)$ is gathered for each resulting edge e. In the last stage, based on push/pop sequences, backward traversal of the WTG is performed to determine the correct target nodes of edges that have not been resolved yet.

Algorithm 4.1: ConstructInitialEdges

```
1  foreach $w \in$ Win do
2      foreach widget event $[v,t]$ with callback $[v,h]$ for $w$ do
3          if $MayOpenNone([v,h])$ then
4              AddEdge($w, w, [v,t]$)
5          foreach $open(w') \in Open([v,h])$ do
6              AddEdge($w, w', [v,t]$)
7      if $w$ is an activity $a$ with options menu $m$ then
8          AddEdge($a, m, [a, menu]$)
9      AddEdge($w, w, [w, back]$)
10 foreach menu and dialog $w \in$ Win do
11     FindOwner($w$)
12 foreach window $w \in$ Win do
13     if $w$ is an activity $a$ then
14         AddEdge($a, a, [a, rotate]$)
15         AddEdge($a, a, [a, home]$)
16         AddEdge($a, a, [a, power]$)
17     if $w$ is an options menu $m$ with owner $a$ then
18         AddEdge($m, m, [m, rotate]$)
19         AddEdge($m, a, [m, home]$)
20         AddEdge($m, a, [m, power]$)
21     if $w$ is a context menu $m$ with owner $a$ then
22         ...
23     if $w$ is a dialog $d$ with owner $a$ then
24         ...
25 foreach statement $stmt \in post\_operations$ do
26     foreach widget $rcv \in$ GetReceivers($stmt$) do
27         if $rcv \in$ Win then
28             $src\_windows \leftarrow \{rcv\}$
29         else
30             $src\_windows \leftarrow$ GetWindow($rcv$)
31         foreach callback $run\_cb \in$ GetRunCallbacks($stmt$) do
32             if $MayOpenNone([none, run\_cb])$ then
33                 foreach window $src\_window \in src\_windows$ do
34                     AddEdge($src\_window, src\_window, post$)
35             foreach $open(w') \in Open([none, run\_cb])$ do
36                 foreach window $src\_window \in src\_windows$ do
37                     AddEdge($src\_window, w', post$)
```

Algorithm 4.1 is an extension of Algorithm 3.1 to augment the WTG with representation of post events. Edges for such events are constructed through the statements at lines 25–37. Function GetReceivers at line 26 returns the points-to set of the receiver variable for the processed statement, obtained using a flow graph defined in prior work [45]. According to the definitions in Section 4.1.1, the source node of a edge could be (1) an activity on which `runOnUiThread` is called, or (2) the windows containing widgets that are receivers of `post` or `postDelayed`. Lines 27–30 in the algorithm consider these two cases. The post events are associated with callbacks which are the `run` methods of related runnable objects. Helper function GetRunCallbacks looks up the points-to set of the call site's argument to identify the related runnable objects and their corresponding `run` methods. If no object is detected during this process (due to unsound modeling of certain Android features), all possible subtypes of `Runnable` are considered. Unlike widget events, the post events are not triggered by interacting with GUI widgets, and thus no contextual information is provided to analyze the `run` callbacks at line 32 and 35 (as indicated by context *none*). In the case when `run` may complete without opening a window, self edges will be created (line 34). Otherwise, edges from source windows to targets are added at line 37. For the two running example, the edges created due to this processing are shown as e_7 in Figure 4.2b and e_1 in Figure 4.4. The edges for post operations, along with other transitions generated by this stage, are processed by the subsequent stages of the analysis. During these stages, additional edges are generated based on edges created by the first stage. This subsequent processing treats edges for post events the same way it treats the "normal" edges described in the previous chapter, by analyzing `run` methods as if they were event handlers, and establishing callback sequences and push/pop sequences as expected.

4.3 Post Operations through `Handler` Objects

Section 4.1 describes a mechanism that provides asynchronous post operations for background tasks to communicate with the UI thread. The Android framework defines another, more general solution to manage thread interactions. A `Handler` object allows sending and processing instances of `Message` or `Runnable` associated with a thread. When a `Handler` is initialized, it is bound to the thread creating it. By posting `Message` or `Runnable` through a `Handler`, the associated or designated thread will execute the tasks at some point in the future. `Handler` manages task interactions through two rather different mechanisms. The first one is `Message`. Based on specified attributes, a `Message` object can be sent through `Handler` to a designated thread triggering specific tasks. The second strategy is very similar to what is introduced in Section 4.1: a `Handler` allows posting a runnable task to the event queue of the associated thread.

Because modeling `Message` instances and their content requires sophisticated static analysis, in this section we focus only on analyzing the post events that can be triggered through a `Handler`. Our goal is to represent such operations on the existing WTG.

Figure 4.5 illustrates an example based on APV. By entering a valid page number through the options menu item `gotoPageMenuItem`, the user can directly jump to a page of PDF to start reading (lines 23–29). When the page is opened, a zoom-in button residing at the bottom of this page shows up, and disappears after 7 seconds.

```
1  class OpenFileActivity extends Activity {
2    Button zoomButton = ...;
3    Handler zoomHandler = ...;
4    Runnable zoomRunnable = ...;
5    public void onCreate(Bundle savedInstanceState) {
6      RelativeLayout activityLayout = new RelativeLayout(this);
7      this.zoomButton = new Button(this);
8      activityLayout.addView(this.zoomButton);
9      this.zoomHandler = new Handler();
10     this.zoomRunnable = new Runnable() {
11       public void run() {
12         this.zoomButton.setVisibility(View.GONE);
13       }
14     };
15     this.setContentView(activityLayout);
16   }
17   public boolean onOptionsItemSelected(MenuItem menuItem) {
18     if (menuItem == this.gotoPageMenuItem) {
19       final Dialog dialog = new Dialog(this);
20       LinearLayout contents = new LinearLayout(this);
21       Button gotoPageButton = new Button(this);
22       gotoPageButton.setOnClickListener(new OnClickListener() {
23         public void onClick(View v) {
24           int pagecount = ...;
25           dialog.dismiss();
26           ... // open specific PDF page at pagecount
27           this.zoomButton.setVisibility(View.VISIBLE);
28           this.zoomHandler.postDelayed(zoomRunnable,7000);
29         }
30       });
31       contents.addView(gotoPageButton,...);
32       dialog.setContentView(contents);
33       dialog.show();
34     }
35   }
```

Figure 4.5: Example of a `Handler`, derived from APV

This animation is implemented using `Handler` to post `zoomRunnable` into the event queue of the UI thread (line 28). `zoomRunnable` overrides the method `run` at line 12 to disable the visibility of `zoomButton` after certain delay. Note that in this case the UI thread posts the runnable task on itself. As stated in the relevant Android documentation [20], "there are two main uses for a `Handler`: (1) to schedule messages and runnables to be executed as some point in the future; and (2) to enqueue an action to be performed on a different thread than your own.". The APV example illustrates the first use.

In our analysis we focus on the following three standard APIs calls defined in class `Handler`.

- `Handler.post(Runnable)`: This API call enqueues the runnable task on the event queue of the thread associated with the `Handler`. The posted task will be executed in the future when the associated thread is free.

- `Handler.postDelayed(Runnable,long)`: Calling this method will add the runnable task into the event queue. The task will be run on the associated thread once the specified amount of time elapses.

- `Handler.postAtTime(Runnable,long)`: Similarly to `postDelayed`, this API call adds a runnable action to the event queue at a specific time. The task will be run on the associated thread when it is available.

Source window. Similar to other post operations introduced earlier, the post events of a `Handler` may trigger transitions in the WTG. As explained previously, a `Handler` posts the operations into the event queue of the thread that allocates it. We aim to recognize `Handler` instances that are created by lifecycle callbacks or GUI event handler callbacks, since these callbacks are executed by the UI thread and can be directly associated with GUI elements. Specifically, consider a window $w \in \mathbf{Win}$ whose GUI hierarchy involves widgets $v \in \mathbf{View}$, and any callback c triggered by event $e = [v,t]$ where t is an event type (e.g., v could be a button and t could be "click"). If the creation of a `Handler` object is reachable from c in the ICFG, the post events associated with this handler are represented in the WTG by edges starting from node w.

Algorithm 4.2: ConstructInitialEdges

```
1  $source\_windows \leftarrow \emptyset$
2  foreach handler $h$ allocation statement $stmt$ do
3      foreach event or lifecycle callback registered on $v$ and transitively invokes $stmt$ do
4          $src\_windows \leftarrow \emptyset$
5          if $v \in$ Win then
6              $src\_windows \leftarrow src\_windows \cup \{v\}$
7          else
8              $src\_windows \leftarrow src\_windows \cup$ GetWindow($v$)
9      foreach $src\_window \in src\_windows$ do
10         $source\_windows \leftarrow source\_windows \cup \{h \rightarrow src\_window\}$
11 foreach statement $stmt \in handler\_post\_operations$ do
12     foreach callback $run\_cb \in$ GetRunCallbacks($stmt$) do
13         if MayOpenNone([none,run_cb]) then
14             foreach window $src\_window \in$ GetSourceWindows($source\_windows$, $stmt$)
               do
15                 AddEdge($src\_window$, $src\_window$, $post$)
16         foreach $open(w') \in Open([none, run\_cb])$ do
17             foreach window $src\_window \in$ GetSourceWindows($source\_windows$, $stmt$)
               do
18                 AddEdge($src\_window$, $w'$, $post$)
```

Algorithm 4.2 presents the analysis for building stage-1 edges for the post operations through `Handler` objects. As another component of our extensions to handle asynchronous control flow, this algorithm is appended to Algorithm 4.1. The first part of the algorithm (lines 2–10) identifies the source windows for the allocations of `Handler`. For each callback registered on widget v, if it reaches the statement allocating `Handler` h, then the sources are the windows containing v in their GUI hierarchy (line 3–8). Helper function GetWindow returns the windows whose hierarchy structure, computed by prior work [45], involves widget v. Such relationship is recorded in multimap `source_windows` mapping a `Handler` to multiple source windows. The second part of the algorithm (line 11–18) considers each post operation

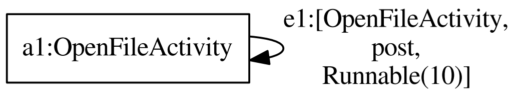


Figure 4.6: WTG for Example 4.5

done through a `Handler`, and analyzes the corresponding `Runnable` object retrieved by method GetRunCallbacks. This method traverses backward the flow graph from [45] (similarly to earlier descriptions of how this flow graph is used to determine points-to sets) and collects a set of allocation expressions of `Runnable` that flow to the corresponding argument of the post operation (line 12). For each `Runnable` object overriding method `run`, the callback analysis introduced in Chapter 2 is used to detect the possible triggered targets, and to build corresponding WTG edges (lines 13–18). Method GetSourceWindows retrieves the associated WTG source nodes from multimap `source_windows`.

The WTG for Figure 4.5 after considering post operations through `Handler`s is presented in Figure 4.6. The operation does not trigger any statement opening or closing windows, thus is represented as a self edge from `OpenFileActivity`. This edge matches the behavior of the run-time execution and correctly captures a window transition missing from the WTG described in the previous chapter.

4.4 Experimental Evaluation

The evaluation of the proposed analysis was conducted for the 20 applications used in the previous chapters. The first goal of this experimental evaluation is to characterize how widely post operations are being used. The second goal is to evaluate the differences between the WTGs from the previous chapter and from our new

Table 4.1: Number of post operations: (1) Activity.runOnUiThread (2) View. post (3) View.postDelayed (4) Handler.post (5) Handler.postDelayed (6) Handler.postAtTime

Application	(1)	(2)	(3)	(4)	(5)	(6)	Total	Resolved
`APV`	5	1	0	0	2	0	8	8
`Astrid`	48	3	4	10	3	0	68	31
`BarcodeScanner`	1	0	0	0	0	0	1	1
`Beem`	3	0	0	6	0	0	9	4
`ConnectBot`	1	0	2	0	2	0	5	2
`FBReader`	38	1	2	0	0	0	41	39
`K9`	17	0	0	17	0	0	34	28
`KeePassDroid`	0	1	0	6	0	0	7	5
`Mileage`	0	0	0	0	0	0	0	0
`MyTracks`	35	0	0	4	3	0	42	17
`NotePad`	0	0	0	0	1	0	1	0
`NPR`	0	1	0	4	4	0	9	4
`OpenManager`	0	0	0	1	0	0	1	0
`OpenSudoku`	0	0	0	2	3	2	7	7
`SipDroid`	0	0	0	0	0	0	0	0
`SuperGenPass`	0	0	0	0	0	0	0	0
`TippyTipper`	0	0	0	2	2	0	4	0
`VLC`	0	1	1	1	4	0	7	2
`VuDroid`	1	4	0	0	0	0	5	2
`XBMC`	5	0	0	104	2	0	111	97

approach. Finally, we also perform case studies for more in-depth understanding of the performance of the new analysis.

4.4.1 Occurrences and Resolution of Post Operations

Table 4.1 contains measurements for the six post operations defined in Section 4.1.1 and Section 4.3. Columns (1)–(6) show the numbers of their invocations in the application code. A total of 360 post operations, presented in column "Total", are detected across the 20 applications. Column "Resolved" shows the number of post operations that were successfully resolved by the proposed approach, leading to the creation of WTG edges. A total of 247 post operations (out of 360) were resolved to WTG edges. As described earlier, invocations of `Activity.runOnUiThread`, `View.post` and `View.postDelayed` can be resolved by our analysis if their receiver activity/widget objects and parameter `Runnable` objects can be identified. Similarly, the proposed technique handles `Handler.post`, `Handler.postDelayed` and `Handler.postAtTime` only if their invocations are reachable from an event/lifecycle handling callback, and the associated `Runnable` instances can be successfully identified. Although most of the post operations (69%) are resolved, future work needs to investigate why the rest cannot be resolved. Our case studies, presented later, provide additional examples to illustrate this issue. We also manually examined `Astrid` and `MyTracks` to determine why a large number of post operations cannot be resolved. The reason is that a large number of post operations have empty points-to sets for the receiver. In both cases, the reason is the use of `FragementActivity`, which is an Android feature not modeled in the prior work [45] upon which our WTG construction is built.

4.4.2 Differences Between WTGs

Table 4.2 shows the numbers of WTG edges in three settings for the analysis. Column “Ch.3” shows the number of WTG edges for the approach defined in the previous chapter.[10] As described earlier, some (but not all) of the post operations defined in Section 4.1.1 and Section 4.3 are considered to be part of callback processing. In particular, when these operations are reached during interprocedural control-flow graph traversals, the corresponding method `run` is identified as the callee. The next column, labeled “W/o post”, represents a variation of this analysis where the post operations are ignored during the traversals—that is, the bodies of resolved methods

[10]These measurements were presented in Chapter 3 and are replicated here for convenience.

Table 4.2: Differences between numbers of WTG edges

Application	Ch.3	W/o post	New
APV	105	97	116
Astrid	838	838	993
BarcodeScanner	128	128	129
Beem	132	132	134
ConnectBot	237	237	238
FBReader	41942	5416	5919
K9	516	516	563
KeePassDroid	643	643	2317
Mileage	676	676	676
MyTracks	391	391	555
NotePad	213	213	213
NPR	590	590	598
OpenManager	116	116	116
OpenSudoku	237	237	246
SipDroid	406	406	406
SuperGenPass	64	64	64
TippyTipper	65	65	65
VLC	131	131	136
VuDroid	47	47	51
XBMC	4279	4279	7627

`run` at post operations are not traversed. The last column "New" shows the number of WTG edges when the post operations are analyzed and represented as described earlier in this chapter. As the measurements indicate, the proposed handling of post operations affects the WTGs of 14 out of the 20 programs, with some cases showing significant differences between the two graphs.

4.5 Case Studies

To better understand our experimental results, we considered the six applications used for case studies in the previous chapter. Three of these applications—`OpenManager`, `TippyTipper`, and `VuDroid`—have post operations that cannot be resolved to WTG edges. We studied the reasons for this behavior. Post operations that were resolved occurred in `APV`, `BarcodeScanner`, and `VuDroid`. In those cases, we wanted to evaluate the feasibility of the generated WTG edges. In addition, we also performed a study for `FBReader`, in order to determine why the number of WTG edges is reduced so dramatically.

OpenManager. In order to concisely show an overview of a specific folder, each of the files, along with its thumbnail, is presented as an element of a list view. The only `Handler`, detected by the proposed analysis, is used to create the thumbnails for each file. However, the statements creating `Handler` instances are transitively invoked by the callback `getView` defined in class `ArrayAdapter`. This callback method is neither a widget event handler callback nor a lifecycle callback. Thus, the proposed Algorithm 4.2 cannot capture and represent this post operation in the WTG.

TippyTipper. As a tips calculator, this application provides functionality for splitting a bill. The default number of people sharing the bill can be configured through a preference whose type is `DialogPreference`. When the dialog is displayed, users can long press buttons to either increment or decrement the default number of people. To keep changing the value while users hold the button, developers use a `Handler` to post a `Runnable` object every 0.3 seconds. When the `Runnable` is executed, it increments/decrements the default value by one, then posts itself again using the same `Handler` object until the button is no longer pressed. Similarly to the previous example, our algorithm does not capture the related four edges because the `Handler` initialization statement is transitively invoked by callback `onCreateDialogView`, which is not a callback we consider in the WTG.

VuDroid. In this PDF reader application, three post operations cannot be resolved by our approach, because their associated GUI components are not identified in the solution produced by the prior work from [45]. Four additional edges are created due to the remaining two post operations. For these, the process of decoding a PDF file is offloaded to a background thread to improve UI performance. The post operations are performed after decoding a page to configure parameters of the UI, e.g., screen size. The four WTG edges created for these operations represent valid run-time behaviors; thus, no infeasible WTG edges are produced.

APV. As described in Table 4.1 and 4.2, the proposed analysis resolves all eight post operations, and generates corresponding WTG edges. A total of 14 edges are created.[11] Among these 14 edges, only one does not represent valid run-time execution behavior. In this case, a call to `runOnUiThread` is executed on `OpenFileActivity`; therefore, this activity is considered to be the source node of the corresponding WTG edge. However, in reality, the current window is a dialog owned by the activity and opened by another thread. As a result, the source node of the transition is incorrect. The remaining 13 edges, which have were manually verified, represent transitions that can be observed at run time.

BarcodeScanner. This application scans and processes eleven types of barcodes. The only post operation is detected when an WiFi barcode is read. In this case, a task

[11] Note that in Table 4.2 the difference between columns "W/o post" and "New" for `APV` is larger than 14. The 14 edges for post operations are created in stage 1 of the algorithm; in later stages, additional edges may be created due to these stage-1 edges (e.g., edges for *back* events).

is posted showing a toast to inform the user that a network connection is requested. The proposed analysis creates a self transition for this case, while in the work from the previous chapter this behavior was considered as part of the callback for handling WiFi barcodes. We examined this edge, and verified its feasibility.

FBReader. The functionality of this application relies heavily on post operations and threads. As explained in Section 3.4.1, the analysis from the previous chapter builds a call graph based on the backward traversal of the flow graph, or class hierarchy analysis in case the flow graph does not provide the necessary information. Such resolution is highly imprecise for certain call sites inside two utility methods. From our observations, over 96% of WTG edges are generated because of these two methods. After separating the handling of post operations, the total number of edges is reduced by more than 75%. Through examination of relevant portions of the code, we identified a potential explanation. Figure 4.7 presents a simplified version of the related code. Method `wait` declared at line 45 is one of the problematic utility methods. (The other one has a similar pattern.) In this example, this method is used to start a new thread running the runnable object provided as a parameter. When the thread is initialized at line 46, a backward traversal, starting from the `Runnable` variable `action`, is performed. The reachable instantiated `Runnable` objects will be bound to the thread object. In this case, the `Runnable` instances created at lines 6 and 31 will be coupled with the thread object initialized at line 46. This information is used when the invocation of method `start` at line 46 is resolved: the `run` methods of `Runnable` instances that are associated with this thread are considered as the called target methods. For the analysis proposed in Chapter 3, the statement at line 46 is reached during ICFG traversals from both `BuyBooksActivity` and `NetworkLibraryActivity`.

```
1  public class BuyBooksActivity extends Activity {
2    protected void onCreate(Bundle bundle) {
3      Button okButton = ...;
4      okButton.setOnClickListener(new View.OnClickListener() {
5        public void onClick(View v) {
6          UIUtil.wait(new Runnable() {
7            public void run() {
8              try {
9                downloadBook(...);
10             } catch (ZLNetworkException e) {
11               BuyBooksActivity.this.runOnUiThread(new Runnable() {
12                 public void run() {
13                   new AlertDialog.Builder(BuyBooksActivity.this)
14                     .setPositiveButton(buttonResource.getResource("ok").getValue(), null)
15                     .create().show();
16                 }
17               });
18             }
19           }
20         });
21       }
22     });
23   }
24 }
25 public class ShowBookInfoAction extends BookAction {
26   Activity myActivity = ...;
27   public ShowBookInfoAction(Activity activity) {
28     this.myActivity = activity;
29   }
30   public void run(...) {
31     UIUtil.wait(new Runnable() {
32        public void run() {
33         try {
34           myActivity.runOnUiThread(new Runnable() {
35             public void run() {
36               myActivity.startActivity(new Intent(myActivity, NetworkBookInfoActivity.class));
37             }
38           });
39         } catch (...) {...}
40       });
41     }
42   }
43 }
44 public class UIUtil {
45   public static void wait(Runnable action) {
46       new Thread(action).start();
47   }
48 }
```

Figure 4.7: Studied case derived from FBReader [11]

Consequently, the invocations which trigger window transitions at lines 15 and 36 are explored to generates six edges:

`BuyBooksActivity` → `BuyBooksActivity`

`BuyBooksActivity` → `Dialog`(13)

`BuyBooksActivity` → `NetworkBookInfoActivity`

`NetworkLibraryActivity` → `NetworkLibraryActivity`

`NetworkLibraryActivity` → `Dialog`(13)

`NetworkLibraryActivity` → `NetworkBookInfoActivity`.

While the work from the previous chapter generates all six edges, the proposed generalizations for modeling of post operations eliminate two infeasible edges:

`BuyBooksActivity` → `NetworkBookInfoActivity`

`NetworkLibraryActivity` → `Dialog`(13)

4.6 Limitations

Source of transition may not be associated with the receiver. The proposed analysis is based on the assumption that the receiver of a post operation is (or is related to) the source window of the transition. However, this assumption may not always be true. For example, an infeasible edge is generated in `APV` because our analysis assumes that the current window, which is associated with the receiver of a post operation, is the source node of the WTG edge. However, in the run-time execution, this source node is actually a dialog which references that window, rather than the window itself.

Multiple executions of post operations. Post operations are often used by background threads to communicate with the with UI thread. We have seen cases in which multiple threads, posting post operations, are initialized by a widget event. For example, clicking a button of a dialog starts two threads: one of them executes a post operation to open a new dialog, while another one closes the current dialog. The proposed work represents this scenario by three WTG edges. The first is the edge with a post event opening a new dialog. The second edge is triggered by a post event closing the current dialog. The last one is a self edge for the click button, which neither opens nor closes any windows. However, during run-time execution, such transitions are usually ordered in terms of when they are added into the event queue of the UI thread. The proposed analysis does not capture such ordering constraints.

4.7 Summary

In this chapter we developed a new representation for post operations, to more faithfully describe their run-time behavior. The experimental results show the importance of identifying and analyzing such operations, and also indicate that the GUI behavior is more comprehensively captured, compared to the approach developed in Chapter 3.

CHAPTER 5: 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 [13,47] 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 [9, 16, 36], has similar limitations. As described in Section 2.1.4, 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 [7] 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. This representation does not capture menus/dialogs, does not consider the general GUI effects of event handlers (e.g., window-close) and the triggered callbacks, and does not model the window stack and its state changes. Similar limitations exist for a static/dynamic analysis of UI-based trigger conditions [65], 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 [6] 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. FlowDroid does not model the general propagation of GUI widgets and listeners, nor does it analyze event handlers context sensitively. As discussed in Section 2.1.4, 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 is also not handled. This approach cannot capture the callback sequences described in Table 3.2 and does not consider the state/changes of the window stack. Providing the WTG as input to FlowDroid is an intriguing possibility for future work.

In CHEX [28], 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 [25] also considers all possible permutations of callbacks. AsDroid [22] 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. Apposcopy [12] 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.

An existing work of operational semantics for activities [40] captures aspects of Android control flow, including callbacks and the activity stack, but does not define GUI static models or analysis algorithms. Various other static analyses aim to model the sequences of callbacks in Android, in the context of security analysis (e.g., [12, 22, 25, 28]), GUI model construction (e.g., [62]), race detection (e.g., [26]), leak analysis (e.g., [19, 38]), and static checking (e.g., [39, 64]). None of these techniques provide comprehensive behavior definition/analysis for the key aspects of GUI behavior: widgets, event handlers, callback sequences, and window stack changes. Our work on the WTG develops a more general approach for static analysis and representation of Android GUI behavior, which provides a promising starting point for generalizing existing (and future) static analyses.

Intent analysis has been used extensively to resolve inter-component control flow in Android [7, 9, 12, 13, 22, 35, 36]. While intent resolution is a prerequisite for the static analyses described in this paper, it cannot solve the more general control-flow questions we address. Section 2.2.4 describes the simplified intent analysis used in our work. While conceptually derived from a more complex prior intent analysis [36], our approach only focuses on explicit intents and does not employ general context/flow sensitivity. However, we do consider a limited form of flow sensitivity, by accounting for infeasible CFG edges under a given widget context; other intent analyses do not employ this technique. It is an interesting open question how to properly integrate intent analysis and GUI-based control-flow analysis, since they depend on each other.

GUI models for understanding and testing. Reverse engineering of GUI models has been studied by others (e.g., [17, 30, 32]) and has been applied to Android (e.g., [3, 7, 53, 54, 62]). 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. As results from Chapter 2 indicate, a static approach could produce more comprehensive models—of course, at the expense of potential infeasibility. For the purposes of program understanding, hybrid static/-dynamic techniques are the most promising, and existing work by Yang et al. [62] and Azim et al. [7] 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), dynamic analysis can be made more efficient and complete. Existing examples of such techniques [7, 62] may benefit from our static WTG models, including the path validity check which could be beneficial for dynamic GUI crawling.

Model-based GUI testing. Finite state machines and similar GUI models have been used often as basis for test generation (e.g., [3, 7, 17, 29, 31, 32, 52, 55, 62]). Tests are defined with respect to GUI model coverage criteria (e.g., [31]). An alternative to model-based testing is random testing; for Android, the Monkey tool [34] has been used for this purpose [21]. Other related efforts for testing of Android include the use of concolic execution to create event sequences for Android testing [4, 24], and amplification techniques for testing of exception-handling code [63] and of low responsiveness defects [59].

The WTG models can serve as basis for automated test generation. Several test generation techniques for Android [24, 52, 58, 59] require models of GUI structure and behavior, and the WTG is a promising candidate in such techniques. Some initial results in this direction were presented in Section 3.3. A natural extension for future work is to develop our prior work on generation of leak-exposing test cases [58] to use the WTG.

Leak analysis for Android. Energy-leak defects have been investigated by others, using both static analysis [38] and dynamic analysis [8,27]. In prior work we developed techniques for static detection of memory leaks in Java [57] and for test generation for resource leaks in Android [58]. An interesting direction for future work is to generalize this prior work to (1) perform static data-flow analyses to check for potential leaks along valid WTG paths, and (2) perform test generation to target common patterns of leaks defined in prior work [8, 19, 27, 38, 58].

CHAPTER 6: Conclusions

With the fast growing complexity of software systems, the industry has experienced new challenges in understanding program's behavior, to reveal both performance and functional deficiencies, and to support software development, testing, debugging, optimization and maintenance. These issues are especially important to mobile software due to the limited computing resources on mobile devices, the short development life cycles, and the lack of comprehensive design principles and experienced programmers. Various tools based on static analyses can help to address this challenge. The work presented in this dissertation contributes to a growing foundation of static analyses for Android software.

It is difficult to rely only on dynamic approaches to automatically extract models of behavior for Android software. The ability to trigger certain behaviors (e.g., sequences of events and transitions between GUI windows) requires modeling of complex dependences and contextual information. Thus, static modeling of control flow and data flow is essential for comprehensive understanding, testing, and correctness/performance checking. However, the framework-based and event-driven nature of Android software hides implementation details and introduces a barrier to software understanding and analysis. In particular, the behavior of callbacks can be complicated and varied, with complex interactions between the framework code and application components.

6.1 Contributions

We advance the state of the art in static analysis for Android with the following contributions. Our first contribution (Chapter 2) is a control-flow analysis for callbacks on user-event-driven application components, including activities, menus, dialogs, and their associated widgets. This analysis infers sequencing constraints for callback invocations, which are then encoded in a callback control-flow graph. The key observation that drives the design of this analysis is that a callback should be analyzed separately for different invocation contexts. In particular, an event handling callback may be responsible for handling of GUI events for several different widgets, which may lead to a different behavior for each widget. We design a context-sensitive graph reachability analysis to identify window open/close operations that affect the callback sequences. The context information, which is a static abstraction of a widget, is used to perform several constant propagation analyses (before the main reachability analysis) in order to identify infeasible code under this context. Our experimental evaluation clearly shows that without exploiting such context knowledge, spurious callback sequences may be inferred. In case studies on six applications, we also observed that only few of the inferred callback sequences are infeasible at run time, typically due to imprecision in prior work [45] upon which our analysis is based. These case studies also indicate that there are clear advantages of using static analysis to model the behavior of these applications, since dynamic ripping techniques may miss significant portions of the GUI.

The second contribution of this dissertation (Chapter 3) is based on another important observation: the effects of handling a GUI event depend on the history of prior GUI events and the window open/close operations triggered by them. We formalize this behavior with the help of an abstraction of window stack. Changes to the stack are also associated with interleavings of callbacks that are invoked on the components involved in the stack change. Our careful description of the details of this behavior adds to the body of knowledge in static analysis of Android, and provides a solid foundation for developing control-flow and data-flow analyses. Based on this formalization, we develop a static analysis to construct the window transition graph (WTG), a new program representation suitable as a starting point for developing new static analyses. One can draw an analogy between the WTG and the interprocedural control-flow graph (ICFG) [48], a key representation for traditional static analysis. WTG can be used as a similarly-important static model for Android. In addition to graph generation (which employs the analysis from Chapter 2 as a building block), we define a WTG path validity check. While ICFG path validity is based on proper matching of calls and returns, WTG path validity is based on matching of window open/close operations, as encoded by the evolution of the window stack along the path. Our experimental evaluation highlights the importance of analyzing the history of prior GUI events and window open/close operations when determining the effects of a GUI event. The results also indicate that a significant portion of WTG paths can be statically pruned through the proposed validity check. Case studies on the six applications used in Chapter 2 show that only 6% of the WTG edges are infeasible at run time, which is a positive indicator that highly-precise static GUI models can be constructed automatically.

The third contribution of this dissertation (Chapter 4) considers an important generalization of the WTG: a representation of certain asynchronous operations performed by other threads. We focus on several standard Android mechanisms that allow a background thread to post a runnable task to be executed by the UI thread of the application. The WTG is extended with edges that signify the execution of such posted tasks. Static analysis of the task's code determines the window open/-close operations resulting from task execution. Our measurements indicate that these mechanisms are often used in the analyzed programs. Through several case studies we demonstrate that the generated WTGs are more comprehensive than the ones described in Chapter 3, and that the newly-introduced WTG edges typically represent feasible run-time behavior.

6.2 Future Work

Our experience indicates that existing expertise in static analysis for Android is still lacking in a number of important aspects. First, the semantics of the Android platform is rich and complicated. There are no precise and comprehensive formal descriptions of this semantics. The semantics also changes, in subtle and undocumented ways, as the platform evolves. This presents a significant challenge for developing new analysis algorithms. As one dimension of this problem, it is essential to create a formal description of the important aspects of Android GUI behavior, and to validate it against actual observed run-time executions.

A second challenge is to develop new static analysis abstractions and patterns suitable for modeling Android behavior. As one example, prior work on GUI structural analysis [45] was observed to lack in both comprehensiveness and precision. Similarly, the handling of asynchronous control flow described in Chapter 4 does not cover the full generality of relevant Android mechanisms. We have also observed that several analyses are interdependent: for example, the GUI analyses defined in this dissertation depend on intent analysis, but they also provide control-flow information that is needed by intent analysis. Similarly, the analysis from [45] depends on information about the control flow, but also provides necessary for control-flow analysis. Creating a conceptual framework for an integrated algorithm that is the "cross product" of existing analyses, and evaluating the benefits of such integration, is an important open problem.

While the WTG presents a promising starting point for control-flow analysis, its strengths and deficiencies need to be evaluated further in the context of client data-flow analyses. For example, static information flow analysis [6] could be redefined based on the WTG. Similarly, energy-leak defects observed in prior work [8, 27, 38] could potentially be stated as properties of WTG paths. The same may be possible for other leak-related patterns investigated in earlier work [19, 58]. In addition to static program checking, automated test generation to support existing [24, 52, 58, 59] and future test generation strategies may be able to take advantage of the WTG representation.

BIBLIOGRAPHY

[1] Android dialogs. `http://developer.android.com/guide/topics/ui/dialogs.html`.

[2] Tasks and back stack. `http://developer.android.com/guide/components/tasks-and-back-stack.html`.

[3] 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*, pages 258–261, 2012.

[4] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE*, pages 1–11, 2012.

[5] APV PDF viewer. `code.google.com/p/apv`.

[6] 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*, pages 259–269, 2014.

[7] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA*, pages 641–660, 2013.

[8] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *FSE*, pages 588–598, 2014.

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

[10] P. Dubroy. Memory management for Android applications. In *Google I/O Developers Conference*, 2011.

[11] FBReader e-book reader. `https://fbreader.org/`.

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

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

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

[15] Gator: Program Analysis Toolkit For Android. `web.cse.ohio-state.edu/presto/software/gator`.

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

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

[18] D. Grove and C. Chambers. A framework for call graph construction algorithms. *TOPLAS*, 23(6):685–746, 2001.

[19] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and detecting resource leaks in Android applications. In *ASE*, pages 389–398, 2013.

[20] Handler class documentation. `http://developer.android.com/reference/android/os/Handler.html`.

[21] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *AST*, pages 77–83, 2011.

[22] 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*, pages 1036–1046, 2014.

[23] Intents and intent filters. `developer.android.com/guide/components/intents-filters.html`.

[24] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *ISSTA*, pages 67–77, 2013.

[25] 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*, pages 21–32, 2013.

[26] Y. Lin, C. Radoi, and D. Dig. Retrofitting concurrency for Android applications through refactoring. In *FSE*, pages 341–352, 2014.

[27] Y. Liu, C. Xu, S. C. Cheung, and J. Lu. GreenDroid: Automated diagnosis of energy inefficiency for smartphone applications. *TSE*, 40:911–940, Sept. 2014.

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

[29] A. M. Memon. An event-flow model of GUI-based applications for testing. *STVR*, 17(3):137–157, 2007.

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

[31] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *FSE*, pages 256–267, 2001.

[32] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *TSE*, 31(10):884–896, 2005.

[33] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *TOSEM*, 14(1):1–41, 2005.

[34] Monkey: UI/Application exerciser for Android. `developer.android.com/tools/help/monkey.html`.

[35] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *ICSE*, pages 77–88, 2015.

[36] 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.

[37] OpenManager: File manager for Android. `github.com/nexes/Android-File-Manager`.

[38] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake? In *MobiSys*, pages 267–280, 2012.

[39] E. Payet and F. Spoto. Static analysis of Android programs. *IST*, 54(11):1192–1201, 2012.

[40] E. Payet and F. Spoto. An operational semantics for Android activities. In *PEPM*, pages 121–132, 2014.

[41] M. Prasad. Personal communication, July 2014.

[42] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[43] Robotium testing framework for Android. `code.google.com/p/robotium`.

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

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

[46] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TCS*, 167(1-2):131–170, 1996.

[47] SCanDroid: Security Certifier for anDroid. `spruce.cs.ucr.edu/SCanDroid/tutorial.html`.

[48] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[49] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, pages 17–30, 2011.

[50] *Soot Analysis Framework*. `http://www.sable.mcgill.ca/soot`.

[51] Stopping and restarting an activity. `developer.android.com/training/basics/activity-lifecycle/stopping.html`.

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

[53] P. Tramontana. Android GUI Ripper. `wpage.unina.it/ptramont/GUIRipperWiki.htm`.

[54] P. Wang, B. Liang, W. You, J. Li, and W. Shi. Automatic Android GUI traversal with high coverage. In *CSNT*, pages 1161 – 1166, 2014.

[55] Q. Xie and A. M. Memon. Using a pilot study to derive a GUI model for automated testing. *TOSEM*, 18(2):7:1–7:35, 2008.

[56] D. Yan. *Program Analyses for Understanding the Behavior and Performance of Traditional and Mobile Object-Oriented Software*. PhD thesis, Ohio State University, July 2014.

[57] D. Yan, G. Xu, S. Yang, and A. Rountev. LeakChecker: Practical static memory leak detection for managed languages. In *CGO*, pages 87–97, 2014.

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

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

[60] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *ICSE*, pages 89–99, 2015.

[61] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for Android. In *ASE*, 2015.

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

[63] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *ICSE*, pages 595–605, 2012.

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

[65] 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*, pages 93–104, 2012.