Static Reference Analysis for GUI Objects in Android Software

Atanas Rountev
Ohio State University
rountev@cse.ohio-state.edu

Dacong Yan
Ohio State University
yan.379@osu.edu

ABSTRACT

The popularity of Android software has grown dramatically in the last few years. It is essential for researchers in programming languages and compilers to contribute new techniques in this increasingly important area. Such techniques require a foundation of program analyses for Android. The target of our work is static object reference analysis, which models the flow of object references. Existing reference analyses cannot be applied directly to Android because the software is component-based and event-driven.

An Android application is driven by a graphical user interface (GUI), with GUI objects responding to user actions. These objects and the event handlers associated with them ultimately determine the possible flow of control and data. We propose the first static analysis to model GUI-related Android objects, their flow through the application, and their interactions with each other via the abstractions defined by the Android platform. A formal semantics for the relevant Android constructs is developed to provide a solid foundation for this and other analyses. Next, we propose a constraint-based reference analysis based on the semantics. The analysis employs a constraint graph to model the flow of GUI objects, the hierarchical structure of these objects, and the effects of relevant Android operations. Experimental evaluation on real-world Android applications strongly suggests that the analysis achieves high precision with low cost.

The analysis enables static modeling of control/data flow that is foundational for compiler analyses, instrumentation for event/interaction profiling, static error checking, security analysis, test generation, and automated debugging. It provides a key component to be used by compile-time analysis researchers in the growing area of Android software.

Categories and Subject Descriptors

F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis*

CGO '14, February 15 - 19 2014, Orlando, FL, USA

http://dx.doi.org/10.1145/2544137.2544159

General Terms

Algorithms, experimentation, measurement

Keywords

Android, GUI analysis, reference analysis

1. INTRODUCTION

The popularity of Android software has grown dramatically in the last few years. Android is the most widely used smartphone platform [10]. Tablets and e-book readers based on Android are also becoming popular. The widespread use of these devices poses significant demands on software performance and quality. However, developer expertise and research advances are still immature compared to older areas of computing. It is essential for researchers in programming languages and compilers to contribute new techniques in this increasingly important area of the computing landscape.

The development of automated tools for performance optimization, profiling, error checking, security analysis, and testing/debugging requires a foundation of program analyses for Android. The target of our work is static *object reference analysis*, which models the flow of object references in the program. Reference analysis for object-oriented languages [19] (also referred to as pointer analysis) has been studied extensively due to its essential role as a prerequisite for many other static analyses. For example, interprocedural control-flow analysis for object-oriented software requires information about the object references that can be observed at polymorphic calls. Another typical example is data dependence analysis, which requires object reference information.

Existing reference analyses cannot be applied directly to Android applications. Although the underlying language is Java, Android software is built using a *component-based* approach where the platform manages the lifetime, behavior, and data of application components. Furthermore, the software is *event-driven*: an Android application is driven by a graphical user interface (GUI), with GUI-related objects responding to user actions (e.g., pressing a button). The set of GUI objects and the event handlers associated with them ultimately determine the possible flow of control and data in the application. To the best of our knowledge, at present there does not exist any work on static analyses to model the details of this GUI-driven control/data flow.

Proposed analysis We propose *the first static analysis* to model the set of GUI-related Android objects, their flow through the application, and their interactions with each

other via the abstractions defined by the Android platform. A number of features make this analysis different from a traditional reference analysis. First, the creation of GUI objects is often implicit, based on external declarative information. The correct modeling of this creation is essential for the proposed analysis. Second, there is a hierarchical structure of GUI objects; this structure affects the run-time behavior and must be modeled statically. In addition, GUI objects are typically accessed via object ids, which requires tracking of such ids and modeling the effects of their use. Finally, it is critical to track the association between a GUI object and the event handlers that respond to user actions on this object.

Motivation The creation, propagation, and interactions of GUI-related objects directly affect the application's runtime behavior. They define the control flow, and in particular the possible GUI events that drive the application, and the invocations of handlers for these events. They also determine aspects of the data flow: for example, text entered by the user (e.g., a password) is obtained with the help of a particular GUI object and flows from it, via the event handler, to the rest of the application. Static analysis to model this control/data flow is foundational for compiler analyses, instrumentation for event/interaction profiling, static error checking, security analysis, test generation, and automated debugging. A body of existing work can directly benefit from our analysis, including static error checking (e.g., [28, 18, 17]), run-time exploration for dynamic analyses for profiling, energy analysis, security analysis, responsiveness analysis, and systematic testing (e.g., [7, 16, 23, 1, 25, 24, 12, 2]), static security analysis (e.g., [4, 9, 5, 11, 15, 8]), and reverse engineering (e.g., [26]). These connections are discussed in Section 6.

Contributions The contributions of this work are

- A *formal semantics* for GUI-related Android constructs. This semantics provides a solid foundation for the development of this and other analysis algorithms.
- A *constraint-based static analysis*. The analysis employs a constraint graph to model the flow of GUI-related objects, the hierarchical structure of these objects, and the effects of relevant Android operations.
- An *experimental evaluation* on real-world Android applications. The results strongly suggest that the analysis achieves high precision with low cost.

These contributions provide a key component for an analysis infrastructure to be used by compile-time analysis researchers in the increasingly important area of Android software.

2. BACKGROUND AND EXAMPLE

Figure 1 shows an example derived from `ConnectBot` [6], an SSH client with more than one million installations according to app market statistics. `ConsoleActivity` defines an *activity*. An activity class is an application class that is a direct or transitive subclass of `android.app.Activity`. Activity objects (referred to as "activities") are the core application components, and they are managed by the platform through various callbacks. When an activity is started (by another activity or by an external application), the platform creates the activity object and invokes the `onCreate` callback method defined at lines 8–16.

```
1 class ConsoleActivity extends Activity {
2   ViewFlipper flip;

3   View findCurrentView(int a) {
4     ViewFlipper b = this.flip;
5     View c = b.getCurrentView(); // FindView
6     View d = c.findViewById(a); // FindView
7     return d; }

8   void onCreate() {
9     this.setContentView(R.layout.act_console); // Inflate
10    View e = this.findViewById(R.id.console_flip); // FindView
11    ViewFlipper f = (ViewFlipper) e;
12    this.flip = f;
13    View g = this.findViewById(R.id.button_esc); // FindView
14    ImageView h = (ImageView) g;
15    EscapeButtonListener j = new EscapeButtonListener(this);
16    h.setOnClickListener(j); // SetListener }

17  void addNewTerminalView(TerminalBridge bridge) {
18    LayoutInflater inflater = ... // helper object
19    View k = inflater.inflate(R.layout.item_terminal); // Inflate
20    RelativeLayout m = (RelativeLayout) k;
21    TerminalView n = new TerminalView(bridge);
22    n.setId(R.id.console_flip); // SetId
23    m.addView(n); // AddView
24    ViewFlipper p = this.flip;
25    p.addView(m); // AddView } }

26 class EscapeButtonListener implements OnClickListener {
27  ConsoleActivity cact;

28  EscapeButtonListener(ConsoleActivity q) {
29    this.cact = q; }

30  void onClick(View r) {
31    ConsoleActivity s = this.cact;
32    View t = s.findCurrentView(R.id.console_flip);
33    TerminalView v = (TerminalView) t;
34    // send ESC key to terminal associated with v } }

act_console.xml:
   <RelativeLayout ... >
      <ViewFlipper android:id="@+id/console_flip" ... />
      <RelativeLayout android:id="@+id/keyboard_group" ... >
         <ImageView android:id="@+id/button_esc" ... />
         ...
      </RelativeLayout>
   ...
   </RelativeLayout>

item_terminal.xml
   <RelativeLayout ... >
      <TextView android:id="@+id/terminal_overlay" ... />
   </RelativeLayout>
```

Figure 1: Example based on `ConnectBot` [6].

An activity presents to the user a window with GUI elements, defined with the help of *views*. A view class v is a direct or transitive subclass of `android.view.View`. Instances of such classes represent GUI widgets that can be observed and manipulated by the user (e.g., buttons), as well as logical groups of such objects. A developer may (but does not have to) introduce application-specific view classes. The example uses standard view classes `ViewFlipper`, `ImageView`, and `RelativeLayout`, as well as an application-defined view class `TerminalView`.

`ImageView` displays an image; in this example, it is used to show the icon of an ESC button for an SSH terminal. `RelativeLayout` is a container for a set of children views, and it itself is not directly visible to the user. `ViewFlipper` is a container that can animate between several children views, by flipping between children (e.g., flipping could happen when the user swipes across the touchscreen). `TerminalView` is an application class providing the GUI for an SSH terminal

window; the source code of this class is not shown in the figure. The two XML files `act_console` and `item_terminal` shown at the bottom of the figure define hierarchies of GUI widgets based on this set of classes.

Layouts Line 9 reads a *GUI layout definition* from XML file `act_console` and instantiates a hierarchy of views. Android best practices suggest that the definition of the visual layout be separated from the code. A layout definition describes a hierarchical structure of views. Using standard tools, these definitions are compiled to Java code. For each layout, there is a unique integer id defined by a final static field in an automatically-generated class `R.layout` (e.g., `R.layout.act_console` and `R.layout.item_terminal`). The values of these ids are used as parameters to several *layout inflaters*. A layout inflater is a method that, given a layout id, "inflates" the definition to a view hierarchy. In general, the parameter of an inflater call can be an integer variable that is (transitively) assigned a layout id. In the example, the inflater call at line 9 associates the new hierarchy with the activity, while the inflater call at line 19 just returns the root view.

A layout definition describes a tree. An inner node represents a container view (e.g., `RelativeLayout`), which is a wrapper around children views. The leaf nodes represent basic GUI components (e.g., `ImageView` and `TextView`). Some nodes may have string ids. For each such *view id* in the definition there is a corresponding integer field in class `R.id`: for example, fields `R.id.console_flip`, `R.id.keyboard_group`, and `R.id.button_esc` correspond to the view ids defined in `act_console`.

Operations on views In `onCreate`, lines 10 and 13 contain *find-view* calls. Such calls use a view id to search for a view in a given hierarchy—in this case, the hierarchy associated with the activity. In addition, line 16 contains a *set-listener* call, which associates a listener (i.e., handler) of click events with the `ImageView` representing an ESC button. The handling of a click event (method `onClick` at lines 30–34) will be discussed shortly. Both find-view operations and set-listener operations are modeled by our analysis, in order to represent statically the propagation of views and the control/data flow due to event handlers for these views.

Method `addNewTerminalView` (lines 17–25) updates the GUI when a new SSH terminal is opened. Calls to this method occur in the rest of the code of `ConsoleActivity`; for brevity, this code is not shown in the example. The call at line 19 inflates a layout defined by XML file `item_terminal` and returns the root `RelativeLayout`. Line 21 shows a programmatically-created instance of `TerminalView`. This class (a subclass of `View`) is not defined or instantiated by the Android platform but rather by the application. At line 22, the id is explicitly set through a *set-id* operation, and at line 23 the new terminal view becomes a child of the inflated `RelativeLayout` through an *add-child* operation. At line 25 the `RelativeLayout` becomes a child of the `ViewFlipper` due to another add-child operation. As several SSH terminals are opened, each call to `addNewTerminalView` extends the hierarchy rooted at the `ViewFlipper` with a new subtree of widgets. The flipper allows the user to flip through the multiple terminals, using swiping motions.

This method illustrates several challenges for the proposed static analysis. First, the program can freely mix inflated views and programmatically-created views. Second, the parent-child relationships between views can be established either through inflation or through explicit add-child operations (lines 23 and 25). Similarly, view ids can be set during inflation or via set-id calls (line 22). Changes to children and ids can in turn affect find-view operations. Consider the call at line 32 in event handler `onClick`, which handles a click event on the ESC button defined by view id `button_esc`. This handler calls helper method `findCurrentView(int)`, which queries the flipper about which of its children is currently visible; this is done by the find-view call at line 5. These children are the `RelativeLayout`s created at line 19 and added as children at line 25; the effects of these two operations need to be modeled properly in order to handle correctly the call at line 5. Furthermore, another find-view call at line 6 searches the hierarchy rooted at the `RelativeLayout` for a view with the given id. This behavior is affected by the set-id operation at line 22, the add-child operation at line 23, and the interprocedural propagation of the view id to parameter `a`.

Event handlers GUI objects in the application can be associated with event handlers. Consider the ESC button, defined by view id `button_esc` and represented at run time by an instance of `ImageView`. This instance is created by the inflater call at line 9 and retrieved by the find-view call at line 13. Click events on this GUI widget are handled by the *listener object* created at line 15. This listener is registered with the GUI object through the set-listener call at line 16. The event handling happens in method `onClick` (lines 30–34). This method's signature is defined in interface `android.view.View.OnClickListener` and is used for the callback methods invoked by the Android platform when a click event occurs. A parameter of the callback is the view `r` on which the event occurred—in this case, the `ImageView` for the ESC button. This example illustrates that a static analysis of GUI objects should account for (1) the association between a GUI object and its listener objects, and (2) the implicit flow of the GUI object as a parameter of callbacks to event handler methods.

3. SEMANTICS OF RELEVANT ANDROID CONSTRUCTS

The creation and propagation of views, together with their interactions with activities and listeners, define a critical component of the run-time behavior of Android applications. These interactions affect the control flow—for example, they define the set of possible GUI events at each moment of the execution (based on the available views), the event-handling code for them, and the effects of this handling (e.g., starting new activities, sending data over the network, etc.). The data flow is also directly affected: for example, text entered by the user is associated with a particular view and flows from that view to the corresponding listener, and from there to other components of the application.

Static analysis to model this run-time behavior is highly desirable as foundation for compiler analyses, instrumentation for profiling of GUI-driven events and interactions, static error checking, security analysis, test generation, and automated debugging. (Some of these analyses are discussed in Section 6.) To the best of our knowledge, at present there does not exist such a static analysis. Our goal is to develop semantic foundations and analysis algorithms for solving this problem. We propose a principled solution, starting with a definition of the semantics of relevant Android constructs

$$
\begin{array}{rcll}
\xi_c & \in & \mathit{Loc} & \text{heap locations} \\
\rho & \in & \mathit{Env} = \mathit{Var} \to \mathit{Loc} & \text{environments} \\
\eta & \in & \mathit{Heap} = \mathit{Loc} \times \mathit{Field} \to \mathit{Loc} & \text{heaps}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}[\![x]\!](\rho,\eta) & = & \rho(x) \\
\mathcal{E}[\![x.f]\!](\rho,\eta) & = & \eta(\rho(x),f) \\
\mathcal{E}[\![\texttt{new } c]\!](\rho,\eta) & = & \xi_c \\
\langle x := e,\rho,\eta\rangle & \to & \langle \rho[x \mapsto \mathcal{E}[\![e]\!](\rho,\eta)],\eta\rangle \\
\langle x.f := y,\rho,\eta\rangle & \to & \langle \rho,\eta[(\rho(x),f) \mapsto \rho(y)]\rangle \\
\langle z := x.m(y),\rho,\eta\rangle & \to & \langle (s_m; z := r_m), \\
 & & \rho[\texttt{this}_m \mapsto \rho(x), p_m \mapsto \rho(y)],\eta\rangle
\end{array}
$$

Figure 2: Semantic domains and functions.

(this section) and using it to define a constraint-based analysis for it (next section).

3.1 Syntax and Semantics of JLite

This subsection describes JLite, a subset of Java that contains all essential language features needed to present the proposed reference analysis for Android software.

Syntactic entities A program contains a set of Java classes. ("Class" will be used to refer to both classes and interfaces.) The following syntactic categories are considered: classes $c \in \mathit{Class}$, methods and constructors $m \in \mathit{Method}$, fields $f \in \mathit{Field}$, statements $s \in \mathit{Stmt}$, and locals/formals $x, y, z, p, r \in \mathit{Var}$. Each method m has a name, formal parameters $\texttt{this}_m$ and p_m, and an artificial return variable r_m. Each method's body is a statement s, defined by $s ::= s_1; s_2 \mid x := \texttt{new } c \mid x := y \mid x := y.f \mid x.f := y \mid z := x.m(y)$. In a minor abuse of notation, here c and m denote the name of class c and method m. Since we are interested in an analysis that abstracts away the intraprocedural control flow—as typically done in reference analysis for Java—conditional statements and loops are omitted. Each variable is of reference type. An assignment to return variable r_m represents a possible return value of method m.

Operational semantics Figure 2 shows the domains and functions used to define the semantics of JLite. A heap location ξ_c is labeled with the class c it instantiates. An environment ρ defines values for locals, formals, and return variables; the values are heap locations. For simplicity of presentation, we formulate the semantics in the absence of recursion (i.e., the elements of *Var* provide unique names for stack locations), and we also assume that all stack and heap locations are properly initialized before being read. A heap η represents the values of fields of heap objects.

Semantic function $\mathcal{E} : \mathit{Expr} \to \mathit{Env} \times \mathit{Heap} \to \mathit{Loc}$ provides the meaning of an expression $e \in \mathit{Expr}$. In the rule for $\mathcal{E}[\![\texttt{new } c]\!]$, ξ_c is a new location that does not occur in ρ or η. The standard rules for sequencing ($s ::= s_1; s_2$) are not shown. For a method call, formals $\texttt{this}_m$ and p_m of m obtains their values from the corresponding actuals. The body s_m of m, followed by propagation of m's return variable r_m, are executed in the updated environment. For brevity, the presentation assumes that each call is statically resolved to a unique target method.

3.2 Syntax and Semantics of ALite

Next we describe ALite, an extension of JLite that introduces the relevant Android constructs. An input program contains a set *Class* of Java classes with the syntactic structure described earlier. Some of these classes are application classes, while others are provided by the Android platform. The analysis aims to model explicitly the complex high-level semantics of Android, rather than analyzing the low-level semantics of platform code; thus, the bodies of methods in platform classes are not included in the input program. The following categories are of particular interest: $a \in \mathit{ActivityClass}$, $v \in \mathit{ViewClass}$, and $h \in \mathit{ListenerClass}$. As discussed in Section 2, an activity class a is an application class that is a subclass of `Activity`, and a view class v is a subclass of `View`. A listener class h implements event handlers associated with views, as discussed later.

3.2.1 Layout Definitions

To represent the effects of layout definitions, the syntax of statements can be extended with $s ::= \ldots \mid x := \texttt{R.layout}.f \mid x := \texttt{R.id}.f$ to reflect the occurrences of layout ids and view ids in the code. XML layout information can be abstracted as follows. A set of $id \in \mathbb{Z}$ defines layout ids and view ids. A layout/view id is the integer value of a constant field from class `R.layout/id` (e.g., `0x7f030000`). A node in a layout definition is (v, id) where v is a view class. There could be several nodes that are instances of the same v (e.g., several buttons in a layout). A layout edge shows a parent-child relationship between views. For example, for `act_console` in Figure 1, one of the layout edges is from parent (`RelativeLayout`, `keyboard_group`) to child (`ImageView`, `button_esc`). A layout definition is a set of layout edges that form a rooted tree.

Semantics To express the effects of layout inflation, we first generalize the environment and the heap:

$$
\begin{array}{rcl}
\mathit{Env} & = & \mathit{Var} \to \mathit{Loc} \cup \mathbb{Z} \\
\mathit{Heap} & = & \mathit{Loc} \times \mathit{Field} \to \mathit{Loc} \cup \\
 & & \mathit{View} \times \{\texttt{vid}\} \to \mathbb{Z} \cup \\
 & & \mathit{View} \times \{\texttt{children}\} \to \mathcal{P}(\mathit{View})
\end{array}
$$

The value of a stack location $x \in \mathit{Var}$ can now be a layout/view id. An assignment $x := \texttt{R.layout}.f$ updates $\rho(x)$ with the appropriate layout id (and similarly for view ids).

Set $\mathit{View} \subset \mathit{Loc}$ denotes all instances of all view classes in the heap. An artificial field `vid` for a view refers to the corresponding view id. Another artificial field `children` refers to the set of children views. The effects of inflating a layout can be captured by inflater semantic functions:

$$
\begin{array}{l}
\mathcal{I}_N : \mathbb{Z} \to \mathit{Env} \times \mathit{Heap} \to \mathcal{P}(\mathit{View} \times \mathbb{Z}) \\
\mathcal{I}_E : \mathbb{Z} \to \mathit{Env} \times \mathit{Heap} \to \mathcal{P}(\mathit{View} \times \mathit{View})
\end{array}
$$

For layout id id_l, $\mathcal{I}_N[\![id_l]\!](\rho,\eta)$ is a set of pairs (ξ_v, id), one for each layout node (v, id). Here ξ_v is a new heap location representing an instance of view class v, and id is the corresponding view id. For the inflation of layout edges, $\mathcal{I}_E[\![id_l]\!](\rho,\eta)$ defines pairs (ξ,ξ') that correspond to layout edges. The semantics of an inflater call is

$$
[\textsc{Inflate}_1] \quad \langle z := x.m(y),\rho,\eta\rangle \to \langle \rho[z \mapsto \xi^{root}],\eta'\rangle
$$

where ξ^{root} is the view at the root of the hierarchy, since the return value of the inflater call is that root. In the updated heap η', for each newly-created view ξ, fields `vid` and `children` are initialized based on $\mathcal{I}_N[\![\rho(y)]\!]$ and $\mathcal{I}_E[\![\rho(y)]\!]$. The object referred to by x is a helper object provided by the platform to implement the inflation. For example, at line 19 in Figure 1, variable `inflater` refers to this helper.

For this call, $\mathcal{I}_N[\![\texttt{item\_terminal}]\!]$ produces $(\xi^1, \texttt{no\_id})$ and $(\xi^2, \texttt{terminal\_overlay})$, where ξ^1 is a new instance of `RelativeLayout` and ξ^2 is a new instance of `TextView`. (Here `no_id` is a special value used to denote the absence of a view id.) Using rule INFLATE$_1$, heap η' has $\xi^1.\texttt{children} = \{\xi^2\}$, $\xi^1.\texttt{vid} = \texttt{no\_id}$, and $\xi^2.\texttt{vid} = \texttt{terminal\_overlay}$; in addition, $\rho(\texttt{k}) = \xi^1$.

3.2.2 Operations on Views

Views created through inflation or through explicit instantiation (`new` v) can be subjected to several operations defined by the Android platform. Correct modeling of the semantic effects of these operations is essential for our analysis.

Associations with activities A view can be associated with an activity. When the activity is active, this view and the view hierarchy rooted at it define the GUI content displayed to the user. As discussed shortly, this association allows hierarchy elements to be accessed programmatically through the activity. The relevant operations are as follows. First, inflater method `Activity.setContentView(int)` can be invoked on an activity, with the parameter being the layout id. As a result, the root of the inflated view hierarchy becomes associated with the activity, rather than being returned from the call. The natural generalization of the semantics is $\mathit{Heap} = \ldots \cup \mathit{Activity} \times \{\texttt{root}\} \rightarrow \mathit{View}$ where `root` is an artificial field for the activity. The semantic rules are extended as expected

$$[\text{INFLATE}_2]\langle x.m(y), \rho, \eta\rangle \rightarrow \langle \rho, \eta[\ldots][(\rho(x), \texttt{root}) \mapsto \xi^{root}]\rangle$$

where $\rho(x) \in \mathit{Activity}$ and $[\ldots]$ represents the heap updates due to inflation, similarly to rule INFLATE$_1$. For example, at line 9 in Figure 1, $\rho(\texttt{this})$ is an activity object ξ^1, and $\xi^1.\texttt{root} = \xi^2$ where ξ^2 is the `RelativeLayout` at the root of the new `act_console` layout instance. Note that similar inflation operations exist for objects other than activities (e.g., for dialogs) and can be modeled in the same manner.

A call to `Activity.setContentView(View)` can be used to create an association between an activity and an existing view. The parameter is a view that could come from several sources—for example, it could be programmatically created, or it could be looked up from an inflated view hierarchy. The semantic effects are

$$[\text{ADDVIEW}_1]\langle x.m(y), \rho, \eta\rangle \rightarrow \langle \rho, \eta[(\rho(x), \texttt{root}) \mapsto \rho(y)]\rangle$$

The same approach applies to similar operations on non-activity objects (e.g., dialogs).

Associations with other views The parent-child relationship between two views can be established during layout inflation, as discussed earlier. Another mechanism is to explicitly invoke an add-child operation. Several methods with the name `addView` can be used for this purpose; lines 23 and 25 in Figure 1 show two examples. Abstracting such calls as $x.m(y)$ where x refers to the parent and y refers to the child,

$$[\text{ADDVIEW}_2]\ \langle x.m(y), \rho, \eta\rangle \rightarrow \langle \rho, \eta[(\rho(x), \texttt{children}) \mapsto \{\rho(y)\} \cup \eta(\rho(x), \texttt{children})]\rangle$$

where $\rho(x), \rho(y) \in \mathit{View}$. The platform ensures that the new hierarchy is well-formed—specifically, that the parent-child relation corresponds to a tree and not to a more general graph. For brevity, we do not express these constraints.

Associations with ids A view id is an integer identifier associated with a view during inflation. A similar effect can be achieved by using a set-id operation: a call to method `setId(int)`, as shown at line 22 in Figure 1. We use the following rule for such a call:

$$[\text{SETID}] \quad \langle x.m(y), \rho, \eta\rangle \rightarrow \langle \rho, \eta[(\rho(x), \texttt{vid}) \mapsto \rho(y)]\rangle$$

Associations with event handlers A view can be associated with several listeners, which are instances of classes $h \in \mathit{ListenerClass}$. Each such h implements one or more listener interfaces. For example, `EscapeButtonListener` in Figure 1 implements interface `View.OnClickListener` and defines a handler `onClick` for click events. The listeners attached to a view determine which events will be handled by the view, which in turn defines the possible flow of control in response to user actions. To capture these associations, the semantics is extended with $\mathit{Heap} = \ldots \cup \mathit{View} \times \{\texttt{listeners}\} \rightarrow \mathcal{P}(\mathit{Listener})$ where `listeners` is an artificial field and $\mathit{Listener} \subset \mathit{Loc}$ denotes all instances of all listener classes in the heap. Set-listener operations are calls $x.m(y)$ where x is the view and y is the listener; one example is shown at line 16 in Figure 1. The semantic rule is

$$[\text{SETLISTENER}]\ \langle x.m(y), \rho, \eta\rangle \rightarrow \langle \rho, \eta[(\rho(x), \texttt{listeners}) \mapsto \{\rho(y)\} \cup \eta(\rho(x), \texttt{listeners})]\rangle$$

Retrieval of views The view ids play a key role in find-view operations. `View.findViewById(int)` searches the hierarchy rooted at the view and returns the descendant view with the given id. A similar operation can also be applied to an activity, in which case the activity's entire view hierarchy is searched. The semantics is captured by

$$[\text{FINDVIEW}_1] \quad \langle z := x.m(y), \rho, \eta\rangle \rightarrow \langle \rho[z \mapsto \mathit{find}(\rho(x), \rho(y))], \eta\rangle$$

where $\mathit{find}(\xi^0, \mathit{id}) = \xi^n$ if there exists a sequence $\xi^0, \ldots, \xi^n$ such that $\eta(\xi^n, \texttt{vid}) = \mathit{id}$ and $\xi^{k+1} \in \eta(\xi^k, \texttt{children})$ for all k. When `findViewById` is invoked on an activity,

$$[\text{FINDVIEW}_2] \quad \langle z := x.m(y), \rho, \eta\rangle \rightarrow \langle \rho[z \mapsto \mathit{find}(\eta(\rho(x), \texttt{root}), \rho(y))], \eta\rangle$$

There are also operations that, when invoked on a view, retrieve some descendant view with a particular run-time property. A typical example is method `findFocus()`, which returns the descendant view that currently has focus. Similarly, the call to `getCurrentView()` at line 5 of Figure 1 returns the child view that is currently visible. We represent such operations by

$$[\text{FINDVIEW}_3]\ \langle z := x.m(), \rho, \eta\rangle \rightarrow \langle \rho[z \mapsto \mathit{find}_m(\rho(x))], \eta\rangle$$

where function find_m abstracts the specifics of m's run-time behavior when invoked on view $\rho(x)$.

Effects of callbacks The Android platform interacts with the application classes through various callback methods. One typical example is method `onCreate` (lines 8–16 in Figure 1), which is invoked on a `ConsoleActivity` object by the platform code that manages the activity lifecycle. Another example is callback method `onClick` (lines 30–34), which is invoked to handle a click event. The general problem of handling such callbacks in static analysis for Android is challenging. While some techniques have been considered in prior work (e.g., [9, 8]), at present there does not exist a fully comprehensive and precise solution. In our work we do not attempt to model all callbacks or their possible or-

derings: instead, we focus on two important categories that directly affect GUI-related behavior. First, for an activity class a, the implicit creation of an instance of a can be modeled by $t :=$ `new` a. Any Android-defined callback to an application method m on an instance of a can be modeled as a call $t.m()$. For the example, we conceptually extend the program with `t:=new ConsoleActivity` and `t.onCreate()`, which is similar to the approach from [8]. In addition to this modeling of activities, we also model the effects of callbacks to handler methods for GUI events. This modeling is conceptually equivalent to creating additional statements, one per set-listener call. Recall that for a set-listener call $x.m(y)$, x refers to a view and y refers to a listener. The declared type of variable y and the signature of m determine the type of GUI event being handled. Let n be the Android-defined signature of handlers for this event. The callback to the handler can be modeled as $y.n(x)$. For the running example, set-listener call `h.setOnClickListener(j)` at line 16 corresponds to an additional statement `j.onClick(h)`.

4. STATIC REFERENCE ANALYSIS

Given the abstracted language ALITE, we aim to develop a static analysis of the creation and propagation of views, as well as their interactions with activities, listeners, and other views. Specifically, the analysis

- defines static abstractions of run-time objects: views, activities, and listeners
- models the flow of (references to) such objects to stack variables and object fields
- determines the relevant structural relationships, including (1) associations of views with activities and listeners, and (2) parent-child relationships between views

A similar problem for the plain-Java language JLITE can be solved using standard existing techniques. We consider one such solution, based on the construction and analysis of a *constraint graph*. A graph node corresponds to $x \in Var$ (a variable node), $f \in Field$ (a field node), or an expression `new` c (an allocation node; the set of these expressions will be denoted by *Alloc*). Edges represent constraints on the flow of values. For example, an assignment $x := y$ is mapped to an edge $y \to x$, to encode the constraint that any value that flows to y also flows to x. Similarly, $x :=$ `new` c is mapped to `new` $c \to x$ to represent the constraint that `new` c is among the values that flow to x. Reachability from an allocation node determines all locations to which references to the corresponding run-time objects can flow. Such an analysis is usually referred to as a control-flow/calling-context-insensitive, field-based reference analysis [19, 13], and is the starting point for our analysis for Android. Various refinements of this technique have been investigated (e.g., [13, 21, 20]); our analysis developments for Android are orthogonal to these refinements and can be combined with them.

4.1 Constraint Graph

Figure 3 shows several constraint graph nodes and edges for the running example. Some of the nodes have subscripts referring to the line number from Figure 1 where the corresponding element occurs for the first time. Additional nodes and edges are shown in Figure 4; gray nodes represent views.

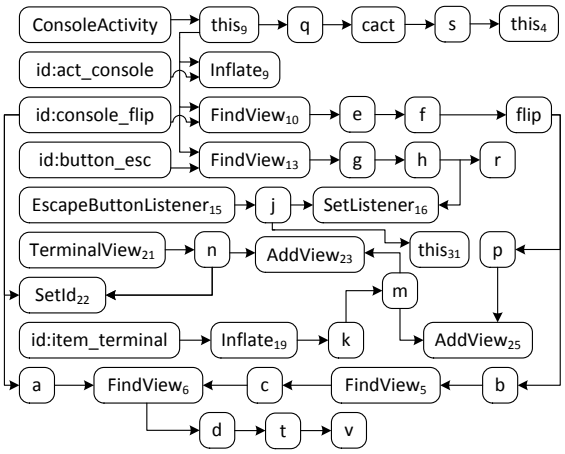

Figure 3: Partial constraint graph for the running example.

Nodes For every integer value in `R.layout`, there is a layout id node $id_l \in LayoutId$. Similarly, a view id node $id_v \in ViewId$ corresponds to each value from `R.id`. Next, an activity node $act \in Activity$ is created for each activity class, to represent instances of this class created implicitly by the Android platform (such instances are never created by `new` in the application code). Four id nodes, as well as the activity node for `ConsoleActivity`, are shown in Figure 3.

A view inflation node $view_{infl} \in ViewInfl$ is introduced for each layout node from XML layouts. This node represents the view created during inflation—that is, the heap object ξ_v created for a layout node (v, id), as defined by rules INFLATE$_{1,2}$. If the same layout is inflated in several places in the application, a "fresh" set of graph nodes is introduced at each inflation site. Six view inflation nodes are illustrated in Figure 4; a subscript $x.y$ refers to the y-th object inflated at line x from Figure 1. We also distinguish the subset of allocation nodes $ViewAlloc \subset Alloc$ that instantiate view classes, and use $view_{alloc}$ to denote such nodes. Similarly, let $Listener \subset Alloc$ be the subset of allocation nodes that instantiate listener classes; elements of this set are denoted by lst. In general, any object could be a listener, including activities and views. To simplify the presentation we assume that activities and views are not listeners, but our implementation handles the general case.

The flow of nodes $view \in View = ViewInfl \cup ViewAlloc$ and the associations of such nodes with act and lst nodes are the core concern of the analysis. This requires modeling of the operations described earlier. For each call $z := x.m(y)$ corresponding to one of the semantic rules, an operation node $op \in Op$ is added to the graph, and the nodes for variables x, y, and z are connected to it. For example, for the find-view operation `d=c.findViewById(a)` at line 6, the graph contains a *FindView* node with incoming edges from `c` and `a`, and an outgoing edge to `d` (shown in Figure 3).

Edges In addition to the JLITE-based edges described earlier, the constraint graph contains edges for Android features. An assignment $x :=$ `R.layout`$.f$ results in an edge $id_l \to x$ from the corresponding layout id node to the variable node x. Similar edges are added for view id nodes id_v. For an activity node act, an edge is added from it to all `this`$_m$ variable nodes, where m is a callback method that could be invoked by the framework with this activity as the

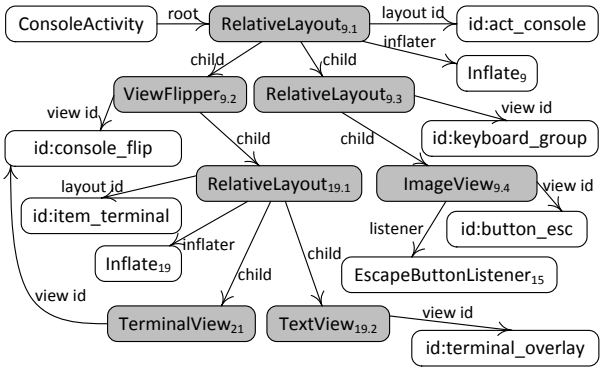

Figure 4: Additional graph nodes and edges.

receiver object. For example, in Figure 3 there is an edge from the activity node to parameter `this`$_9$ of `onCreate`.

All edges described so far model the flow of values. We also use edges $n \Rightarrow n'$ to represent constraints on other relevant relationships. For example, an edge $view_1 \Rightarrow view_2$ between two view nodes shows a parent-child relationship—that is, the constraint $view_2 \in view_1.\texttt{children}$. An edge $view \Rightarrow id_v$ indicates that the view is associated with this view id (by rules InflateId$_{1,2}$ and SetId). An edge $view_{infl} \Rightarrow id_l$ connects the root of an inflated hierarchy with the layout id of the layout that was inflated. Similarly, $view_{infl} \Rightarrow Inflate_{1,2}$ is introduced when the view is the root of the hierarchy inflated by this *Inflate* operation node. An edge $act \Rightarrow view$ indicates that the view is the root of the hierarchy associated with the activity (as set up by rules Inflate$_2$ and AddView$_1$). Finally, $view \Rightarrow lst$ shows that the view is associated with this listener because of rule SetListener. All these categories of edges are illustrated in Figure 4, with edge labels added for clarity.

4.2 Constraint-Based Analysis

We define the analysis in terms of constraints over the nodes and edges of the graph, with the help of two binary relations. First, $ancestorOf \subseteq View \times View$ is the transitive closure of the parent-child relation: $view_1$ $ancestorOf$ $view_2$ if and only if there exists a path in the constraint graph starting at $view_1$, ending at $view_2$, and containing only view nodes and $\Rightarrow$ edges. The second relation is $flowsTo \subseteq (View \cup LayoutId \cup ViewId \cup Activity \cup Listener) \times (Var \cup Field \cup Op)$. This relation shows that the value represented by the first node—a view, an id, an activity, or a listener—flows to the variable, field, or operation represented by the second node. Both relations can grow during the analysis. For example, when two views flow to an *AddView* operation node (corresponding to rule AddView$_2$), a new parent-child edge is added to the constraint graph, which in turn affects $ancestorOf$. The basic inference rules for these two relations are as follows:

$$\frac{n_1 \in View \cup LayoutId \cup ViewId \cup Activity \cup Listener \quad n_2 \in Var \quad n_1 \rightarrow n_2}{n_1 \;\; flowsTo \;\; n_2}$$

$$\frac{n_2 \in Var \cup Field \quad n_3 \in Var \cup Field \cup Op \quad n_2 \rightarrow n_3 \quad n_1 \;\; flowsTo \;\; n_2}{n_1 \;\; flowsTo \;\; n_3}$$

$$\frac{view \in View}{view \;\; ancestorOf \;\; view}$$

$$\frac{view_1 \;\; ancestorOf \;\; view_2 \quad view_2 \Rightarrow view_3}{view_1 \;\; ancestorOf \;\; view_3}$$

For example, in Figure 3, view id `console_flip` flows to operation node $FindView_6$ via variable node `a`, and view `TerminalView`$_{21}$ flows to $SetId_{22}$ and $AddView_{23}$ via `n`. Considering the parent-child edges in Figure 4, the root node `RelativeLayout`$_{9.1}$ is an ancestor of seven nodes.

The inference rules for the semantic rules are described below. For example, for AddView$_1$ we have

$$\frac{act \;\; flowsTo \;\; AddView1 \quad view \;\; flowsTo \;\; AddView1}{act \Rightarrow view}$$

where act is the activity node. Similarly,

$$\frac{view_1 \;\; flowsTo \;\; AddView2 \quad view_2 \;\; flowsTo \;\; AddView2}{view_1 \Rightarrow view_2}$$

assuming that $view_1$ flows to the operation node $AddView_2$ in the role of the parent. For example, `TerminalView`$_{21}$ flows to $AddView_{23}$ in the role of the child (Figure 3). As described shortly, `RelativeLayout`$_{19.1}$ flows to this operation in the role of the parent, via `k` and `m`. As a result, a parent-child edge `RelativeLayout`$_{19.1} \Rightarrow$ `TerminalView`$_{21}$ is created by the analysis, as shown in Figure 4.

For semantic rules SetId and SetListener we have

$$\frac{view \;\; flowsTo \;\; SetId \quad id_v \;\; flowsTo \;\; SetId}{view \Rightarrow id_v}$$

$$\frac{view \;\; flowsTo \;\; SetListener \quad lst \;\; flowsTo \;\; SetListener}{view \Rightarrow lst}$$

In Figure 3, both `TerminalView`$_{21}$ and `console_flip` flow to $SetId_{22}$. This leads to the creation of `TerminalView`$_{21} \Rightarrow$ `console_flip` (shown in Figure 4), which in turn affects relation $ancestorOf$ and the find-view operations.

For rules FindView$_{1,2,3}$ the constraints are

$$\frac{\begin{array}{c} view_1 \;\; flowsTo \;\; FindView1 \quad id_v \;\; flowsTo \;\; FindView1 \\ FindView1 \rightarrow n \quad view_1 \;\; ancestorOf \;\; view_2 \quad view_2 \Rightarrow id_v \end{array}}{view_2 \;\; flowsTo \;\; n}$$

$$\frac{\begin{array}{c} act \;\; flowsTo \;\; FindView2 \quad id_v \;\; flowsTo \;\; FindView2 \\ FindView2 \rightarrow n \quad act \Rightarrow view_1 \\ view_1 \;\; ancestorOf \;\; view_2 \quad view_2 \Rightarrow id_v \end{array}}{view_2 \;\; flowsTo \;\; n}$$

$$\frac{\begin{array}{c} view_1 \;\; flowsTo \;\; FindView3 \\ FindView3 \rightarrow n \quad view_1 \;\; ancestorOf \;\; view_2 \end{array}}{view_2 \;\; flowsTo \;\; n}$$

For example, `ConsoleActivity` and id `button_esc` flow to $FindView_{13}$, and the outgoing edge is to variable `g`. Furthermore, `ConsoleActivity` $\Rightarrow$ `RelativeLayout`$_{9.1}$ because this view is the root of the hierarchy inflated by $Inflate_9$ and associated with the activity. This root is an ancestor of `ImageView`$_{9.4}$, which has an edge to the same view id. Thus, the analysis can conclude that `ImageView`$_{9.4}$ $flowsTo$ `g`. Later this is used to determine that the view flows to $SetListener_{16}$.

Recall that semantic rule FindView$_3$ retrieves some descendant view with a particular run-time property. The

static approximation is to assume that any descendant view can be retrieved, as shown in the constraint rule for *FindView*3 operation nodes. Sometimes more restricted semantics applies: for example, for the call to `getCurrentView()` at line 5 in Figure 1, any child view can be retrieved, but not any deeper descendant. Such refinements are not discussed, but they are employed by our implementation.

For rules INFLATE$_{1,2}$, suppose that a layout id id_l flows to an *Inflate* operation node. In that case, the corresponding layout is inflated and its root node is connected with the inflater node and with the layout id (to capture the origin of the inflated hierarchy). The rules are

$$\frac{id_l \;\; flowsTo \;\; Inflate1 \qquad Inflate1 \rightarrow n \qquad view \Rightarrow Inflate1 \qquad view \Rightarrow id_l}{view \;\; flowsTo \;\; n}$$

$$\frac{act \;\; flowsTo \;\; Inflate2 \qquad id_l \;\; flowsTo \;\; Inflate2 \qquad view \Rightarrow Inflate2 \qquad view \Rightarrow id_l}{act \Rightarrow view}$$

In the first case, the root is propagated to the left-hand side variable at the inflater call. For example, $Inflate_{19}$ has an outgoing edge to `k`, and the analysis determines that $\texttt{RelativeLayout}_{19.1}$ flows to `k` (and from there to several other nodes). In the second case, the call associates the activity with the root object: e.g., at $Inflate_9$ an edge $\texttt{ConsoleActivity} \Rightarrow \texttt{RelativeLayout}_{9.1}$ is created.

4.3 Analysis Algorithm and Implementation

To find a solution to the system of constraints, we employ a fixed-point algorithm. First, the analysis creates the constraint graph edges that can be directly inferred from program statements; all edges in Figure 3 fall in this category. All methods in the application code are considered executable and thus analyzed. Polymorphic calls are resolved using class hierarchy information. Calls to application methods result in constraint graph edges that represent parameter passing and return values. The abstracted semantics refers to a small number of broad categories of relevant operations (e.g., ADDVIEW, SETLISTENER, etc.) which in reality correspond to a wide variety of Android APIs. Some of these APIs have semantic variations that are not discussed here, but are handled by our implementation. Occurrences of these APIs in the application code are recognized and modeled appropriately in the flow graph. The effects of callbacks from the Android platform are also modeled at this time, as outlined at the end of Section 3. However, instead of creating explicit statements, the analysis simply adds constraint graph nodes and edges to simulate the corresponding semantic effects.

The next phase of the analysis uses graph reachability to compute relationships that do not depend on operation nodes; examples include *act flowsTo n* and *id flowsTo n*. In addition, propagation paths that connect operation nodes are computed: for example, in Figure 3, path $Inflate_{19} \rightarrow k \rightarrow m \rightarrow AddView_{23}$ propagates the output of one operation node to the input of another, and its endpoints are recorded for later use. Given the reachability information, *Inflate* nodes are processed (based on reaching layout ids) to create inflated view nodes and the parent-child edges for them. Different variations of the inflater semantics are handled as necessary, and edges to represent relevant semantic effects (e.g., the association between an activity and a root GUI object at INFLATE$_2$) are introduced.

In the final phase of the analysis, a fixed-point computation propagates views through the constraint graph based on the computed propagation paths. For each node that has a view as input or output, a set of reaching views is maintained and updated as necessary. Our implementation is based on the Soot analysis framework [22]. Soot's intermediate representation can be constructed either from source code, or from the Dalvik bytecode specific to Android [3, 14]. Certain Android GUI features are not handled by the current implementation (e.g., dialogs and fragments). Another limitation is that native code is not analyzed, although we have not observed native code that creates GUI objects or registers listeners.

5. EXPERIMENTAL EVALUATION

The analysis was evaluated on the 20 Android applications described in Table 1. Almost all programs have been used in prior work [26, 24, 27, 18]. The table shows the number of application classes and methods, as well as the breakdown of constraint graph nodes. These measurements characterize the relevant application features and provide motivation for the proposed analysis of GUI-related behavior; the results of the analysis are presented in Table 2.

Column "ids" shows the number of layout ids (L) followed by the number of view ids (V). Based on the measurements of layout ids as well as inflater nodes (column "Inflate"), it is clear that XML layouts are widely used and must be modeled in a static analysis. Another observation is that the number of view ids is large, and their use must be accounted for in a static analysis, especially because the number of find-view operations where these ids are used (column "FindView") is also rather large.

Column "views" shows the number of inflated (I) and explicitly allocated (A) view nodes. The large number of views implies a complex GUI structure that requires careful modeling (e.g., in order to generate representative input events for profiling). Most views are inflated, but explicitly allocated views are also present in 15 out of the 20 applications. Explicit manipulation of the view hierarchy via add-child operations (column "AddView") occurs in all but four applications. Our analysis was specifically designed to handle all these features. Event handlers (column "listeners") and the associated set-listener operations (column "SetListener") are commonly used by the applications. Static control/data flow analysis for Android must account for the association between views and the event handlers that respond to them.

Table 2 shows the running time of the analysis and measurements of the computed solution. Even for the larger programs, the analysis time is very practical. Column "receivers" shows the average number of view objects that are receivers at operation nodes (e.g., *FindView* and *AddView*2). Smaller numbers imply higher precision, with 1 being the lower bound. For 16 out of the 20 programs, this average is less than 2. Similar observations can be made for column "parameters", which shows the average number of views reaching an *AddView* node as a parameter. (The four "-" entries correspond to programs without such operations.) Column "results" shows how many views, on average, are results (i.e., outputs) from operations such as *FindView*. The averages are less than 2 for all but one application, which implies that the resolution of find-view operations is highly

App	Classes	Methods	Objects and ids				Operations				
			activities	ids (L/V)	views (I/A)	listeners	Inflate	FindView	AddView	SetId	SetListener
APV	68	415	4	3/12	16/21	16	3	8	19	0	17
Astrid	1228	5782	43	95/223	769/46	181	71	435	63	1	188
BarcodeScanner	126	594	9	9/31	61/0	11	9	35	0	0	10
Beem	284	1883	12	17/50	126/0	12	17	52	0	0	26
ConnectBot	371	2366	11	19/45	140/7	26	20	63	5	1	30
FBReader	954	5452	31	23/111	201/9	43	22	118	13	0	42
K9	815	5311	33	35/153	385/8	54	39	218	11	2	91
KeePassDroid	465	2784	23	19/70	213/12	29	22	103	20	0	29
Mileage	221	1223	64	33/58	335/0	30	37	104	13	6	31
MyTracks	485	2680	35	25/118	2040/4	30	16	110	2	0	28
NPR	249	1359	15	19/88	274/9	17	20	149	22	0	41
NotePad	89	394	8	7/12	18/4	9	7	16	2	0	10
OpenManager	60	252	8	8/46	147/0	20	11	48	0	0	21
OpenSudoku	140	726	10	15/31	109/6	16	16	65	5	3	17
SipDroid	331	2863	12	6/36	75/4	11	6	42	3	0	12
SuperGenPass	65	268	3	4/9	37/0	12	4	11	0	0	11
TippyTipper	57	241	6	6/42	143/22	27	6	43	20	0	30
VLC	242	1374	10	35/91	264/11	45	38	139	9	1	59
VuDroid	69	385	5	2/3	11/6	4	3	5	4	0	4
XBMC	568	3012	24	28/151	467/23	88	27	249	14	0	69

Table 1: Analyzed applications and relevant constraint graph nodes.

App	Time	Average number			
		receivers	parameters	results	listeners
APV	0.39	1.00	1.00	1.44	1.00
Astrid	4.92	3.55	3.09	1.84	1.44
BarcodeScanner	0.65	1.24	-	1.11	1.07
Beem	1.17	1.04	-	1.08	1.00
ConnectBot	1.21	1.00	1.20	1.03	1.04
FBReader	3.28	1.54	1.25	1.65	1.23
K9	4.30	1.15	1.09	1.15	1.00
KeePassDroid	2.09	1.80	1.00	1.73	1.00
Mileage	0.41	2.55	1.38	2.09	1.47
MyTracks	1.55	1.12	1.00	1.92	1.24
NPR	0.87	1.89	1.00	1.66	4.94
NotePad	0.63	1.00	1.00	1.00	1.00
OpenManager	0.39	1.31	-	1.31	1.43
OpenSudoku	0.66	1.10	1.40	1.06	1.25
SipDroid	0.88	1.00	1.00	1.00	1.67
SuperGenPass	0.31	2.17	-	1.69	1.80
TippyTipper	0.18	1.15	1.00	1.00	1.34
VLC	1.15	1.13	1.11	1.14	1.00
VuDroid	0.30	1.00	1.00	1.00	1.00
XBMC	1.74	8.81	2.54	1.86	1.73

Table 2: Analysis running time (in seconds) and average number of objects in the solution for operation nodes.

precise. Finally, column "listeners" shows how many listener objects, on average, are associated with a view object at a set-listener operation. As with the other measurements, this number is typically small, indicating good precision.

In a further case study, we investigated the solutions for four programs: `APV`, `BarcodeScanner`, `SuperGenPass`, and `XBMC`. The first three were chosen because they are the easiest to comprehend (i.e., with the smallest number of views) and have non-singleton solution sets. `XBMC` was chosen because it is an outlier for column "receivers". By manually reasoning about the application and all solutions sets, we determined which solution elements are part of the "perfectly-precise" static solution (i.e., the solution capturing all and only possible run-time behaviors). For `APV`, `BarcodeScanner`, and `SuperGenPass` our solution achieves perfect precision. For `XBMC`, the perfectly-precise measurements would be 3.59 for "receivers", 1.63 for "results", and unchanged for the other two columns. The imprecision is due to the calling-context-insensitive nature of the analysis; applying existing techniques for context sensitivity (e.g., [21, 20]) would lead to an even more precise solution for this outlier.

The evaluation strongly suggests that real-world applications commonly use the Android features we aim to model, and that our approach analyzes these features with high precision and low running time. This makes the proposed analysis a promising building block for a variety of other analyses for Android, as described in the next section.

6. RELATED WORK

Early work by Chaudhuri [4] and follow-up work on the SCanDroid security analysis tool [9] formalizes aspects of the semantics and performs control-flow analysis and security permissions analysis. This effort focuses on activities and other Android components (e.g., background services). These components communicate through intents—objects that describe the operation to be performed—and the analysis models these intents and the inter-component control flow based on them. The implementation is evaluated on a number of synthetic examples. This work does not consider what are the GUI objects, events, and handlers that trigger the inter-component flow of control. For example, `ConsoleActivity` from the running example starts several other activities (through intents). This is done in event handlers for GUI objects. These handlers are defined by listener classes, outside of activity classes. To fully model the inter-component flow of control, it is necessary to (1) establish the association between the `ConsoleActivity` and the GUI object, (2) determine the event handlers associated with this GUI object, and (3) identify the new activities started by these handlers. Later work on related security problems [5, 11, 15] has similar limitations.

The recent development of the A^{3}E tool for automated run-time exploration of Android applications [2] takes advantage of SCanDroid's static analysis to achieve high coverage. Such coverage is essential for a variety of dynamic analyses for profiling, energy analysis, security analysis, re-

sponsiveness analysis, and systematic testing (e.g., [7, 16, 23, 1, 25, 24]). The analysis from SCanDroid is used to construct a static activity transition graph, with nodes representing activities and edges showing the possible transitions between them; this graph is then used to drive the run-time exploration. It it unclear how this static analysis approach handles the general case when arbitrary GUI objects are associated with an activity, their handlers (located outside of the activity class) are registered via set-listener calls, and those handlers trigger transitions to new activities. Similar considerations apply to a hybrid static/dynamic analysis of UI-based trigger condition in Android applications [29], where security-sensitive behaviors are triggered dynamically based on a static model of activity transitions. The model construction in this work is incomplete and can benefit from the general solution provided by our approach.

A similar model, in which nodes represent UI screens and edges show transitions based on GUI events, is used as input to an automated test generation approach based on concolic execution [12]. Essential information encoded in the model is the set of tuples (activity a, GUI object v, event e, handler method h), where v is visible when a is active, and event e on v is handled by h. In this work the models are constructed manually; the output of our analysis can be directly used to automate the generation of these tuples.

Yang et al. [26] present a reverse-engineering tool that combines static and dynamic analysis to construct a model of the application's GUI for testing purposes. The static analysis component identifies the objects that can serve as listeners, and determines the view ids of the GUI objects associated with these listeners. The analysis does not model the actual GUI objects (inflated or explicitly created), does not capture the general flow of these objects through the constructs described in Section 3, and does not account for the flow of view ids. Using our analysis, the generality of this reverse-engineering tool can be increased. Similar issues exist in prior work on a static error checker for GUIs [28]. This tool is based on analysis of call paths that lead to operations on GUI objects. The analysis takes into account the objects created through inflation, but does not model precisely the flow of views due to the operations outlined in Section 3. Similar features and limitations can be seen in another static checker for Android [18]. Employing our analysis would lead to improved generality and precision for these checkers.

Understanding GUI objects and their event handlers is essential for various other analyses of Android applications. For example, an existing static detector of energy-related software defects [17] requires control-flow analysis of the possible execution orders of event handlers. In this work, programmer input is needed to specify these orders. Instead, it may be possible to develop an automated approach based on analysis of activities, GUI objects associated with them, and handlers for these objects; our analysis provides the starting point for such an approach. Another relevant example is FlowDroid [8], a precise flow- and context-sensitive taint analysis which performs extensive interprocedural control-flow and data-flow analysis. As part of this approach, the effects of callbacks are modeled by creating a wrapper main method. Our handling of relevant callbacks conceptually follows a similar approach, but without explicitly creating a wrapper. In FlowDroid, placeholder GUI objects that may flow into these callbacks are created in the wrapper method. In addition, XML layout files are examined to identify potential taint sources and connect them with the statements that access them. It does not appear that the tool models the constructs discussed in Section 3 and the corresponding GUI-related flow. This analysis could be complemented by our approach, which would add general modeling and tracking of GUI objects and their event handlers.

7. CONCLUSIONS

Building a foundation of static analyses for Android is essential for new compile-time and run-time techniques and tools in this increasingly important area of computing. We propose the first static analysis to focus on GUI-related Android objects. The analysis defines abstractions of views, activities, and listeners. It models the flow of such objects, the effects of Android operations, and the relevant structural relationships, including associations of views with activities and listeners, and parent-child view relationships. Our constraint-based algorithm exhibits high precision and low cost. This analysis is an important building block for existing and future compiler analyses, profiling techniques, static error checkers, security analyses, and testing approaches.

Acknowledgments

We thank the CGO reviewers for their valuable feedback. This material is based upon work supported by the U.S. National Science Foundation under grants CCF-1017204 and CCF-1319695, and by a Google Faculty Research Award.

8. REFERENCES

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *International Conference on Automated Software Engineering (ASE)*, pages 258–261, 2012.

[2] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013. To appear.

[3] A. Bartel, J. Klein, Y. L. Traon, and M. Monperrus. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis (SOAP)*, 2012.

[4] A. Chaudhuri. Language-based security on Android. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 1–7, 2009.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 239–252, 2011.

[6] ConnectBot: SSH client for Android. `code.google.com/p/connectbot`.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–6, 2010.

[8] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for Android applications. Technical Report TUD-CS-2013-0113, TU Darmstadt, May 2013.

[9] 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.

[10] Gartner, Inc. Press release, 2012. `www.gartner.com/newsroom/id/2237315`.

[11] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[12] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 67–77, 2013.

[13] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC)*, pages 153–169, 2003.

[14] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android applications to Java bytecode. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 6:1–6:11, 2012.

[15] 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 Symposium*, 2013.

[16] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? In *European Conference on Computer Systems (EuroSys)*, pages 29–42, 2012.

[17] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake? In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 267–280, 2012.

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

[19] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction (CC)*, pages 126–137, 2003.

[20] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 17–30, 2011.

[21] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 387–400, 2006.

[22] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction (CC)*, pages 18–34, 2000.

[23] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: Multi-layer profiling of Android applications. In *International Conference on Mobile Computing and Networking (MobiCom)*, pages 137–148, 2012.

[24] D. Yan, S. Yang, and A. Rountev. Systematic testing for resource leaks in Android applications. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 411–420, 2013.

[25] S. Yang, D. Yan, and A. Rountev. Testing for poor responsiveness in Android applications. In *Workshop on Engineering Mobile-Enabled Systems (MOBS)*, pages 1–6, 2013.

[26] W. Yang, M. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 250–265, 2013.

[27] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *International Conference on Software Engineering (ICSE)*, pages 595–605, 2012.

[28] S. Zhang, H. Lü, and M. D. Ernst. Finding errors in multithreaded GUI applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 243–253, 2012.

[29] 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 *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 93–104, 2012.