On the Unsoundness of Static Analysis for Android GUIs

Yan Wang Hailong Zhang Atanas Rountev
Ohio State University, USA

Abstract

Android software presents exciting new challenges for the static analysis community. However, static analyses for Android are typically unsound. This is due to the lack of specification of the Android framework, the continuous evolution of framework features and behavior, and the absence of soundness arguments and studies by program analysis researchers. Our goal is to investigate one important aspect of this problem: the static modeling of control/data flow due to interactions of the user with the application's GUI. We compare the solutions of three existing static analyses—FlowDroid, IccTA, and GATOR—with the actual run-time behavior. Specifically, we observe the run-time sequences of callbacks and their parameters, and match them against the static abstractions provided by these analyses. This study provides new insights into the unsoundness of existing analysis techniques. We conclude with open questions and action items for program analysis researchers working in this increasingly important area.

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

Keywords Android, static analysis, soundness

1. Introduction

Android is the most popular software platform for mobile devices [11] and presents exciting new challenges for the static analysis community. These challenges are both foundational (e.g., how to model general control/data flow) and in specific application areas (e.g., security, performance, energy usage, etc.) However, due to the framework-based, event-driven nature of the Android execution model, there is lack of conceptual clarity on the run-time semantics and how it can be represented by static abstractions. As a result, static analyses for Android are typically *unsound*. The first goal of our work is to highlight the challenges in developing sound static analyses for Android, and to illustrate them with a case study of three analyses [3, 18, 35]. The results of this study indicate that the underlying techniques have several aspects of unsoundness. Based on these results, our second goal is to outline a research agenda for the program analysis community in order to measure, understand, and reduce unsoundness in static analysis of Android software. We hope this initial investigation will motivate other researchers to perform further in-depth studies of analysis soundness for Android.

SOAP'16, June 14, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4385-5/16/06...
http://dx.doi.org/10.1145/2931021.2931026

Challenges for developing sound analyses There is a substantial number of static analyses for Android. Many are motivated by security-related problems; a representative example is the taint analysis in FlowDroid [3], which will be discussed later. Others consider problems such as battery drain (e.g., [25, 33]), leaks (e.g., [14, 37]), GUI exploration (e.g., [4, 36]), and responsiveness (e.g., [19, 32]). In our experience, existing work (including our own work) almost never discusses the soundness of the proposed analysis. One major reason is *the lack of precise specification of Android structure and behavior*. Android applications are built by extending the functionality of the Android framework (e.g., by subclassing of framework classes and overriding framework methods) and/or directly using framework entities (e.g., GUI widgets). The possible interactions between framework code and application code are overwhelmingly complicated. Most static analyses for Android do not analyze the framework code, because treating it as "plain old Java" — and subjecting it to traditional static analyses for Java — would lose much of the high-level framework semantics. Instead, the typical approach is to (silently) select a subset of framework features and to embed a model of their behavior in the analysis.

This is a highly unsatisfactory state of affairs, because the selected features and their precise run-time behavior typically are poorly specified by analysis designers—to a large extent, because the framework itself is poorly specified. For many programming languages, one can rely on a standardized language specification and on formal semantic definitions for "interesting" subsets of the language (e.g., a classic example is Featherweight Java [16]). Such informal or formal semantics can be used to reason about the (un)soundness of a proposed static analysis. In contrast, there is no "Android framework specification". Although various API documents exist, they are not at the level of rigor and completeness that is expected of a language specification. How can an analysis designer reason about soundness if there is no complete definition of framework features and their run-time behaviors?

Another challenge is the *evolution* of the Android framework. Currently there are 23 framework API revisions ("API levels"). Significant new features have been introduced along the way. The run-time behavior of commonly used features is sometimes changed without any documentation. For example, we have observed undocumented changes in the behavior of menu windows and the related callbacks. Combined with the lack of a framework specification, such changes present an obstacle for static analysis designers.

In this context, perhaps one can only hope for a *soundy* static analysis [21]: common features should be handled soundly and more esoteric ones should be under-approximated. This distinction is quite valuable in established languages, where both the language specification and the static analysis expertise have had time to evolve and mature. For Android software, it is still very much unclear what are the dimensions of such algorithm design decisions. A significant body of future work is required to strengthen our understanding of such foundational static analysis questions. This paper aims to make a small contribution in this direction.

Our proposal One key aspect in understanding and eliminating various sources of unsoundness is to *compare the static analysis solution against some concrete run-time executions*. Of course, this idea is not new—static analysis developers often use it to debug their analysis design and implementation. We propose to use an instance of this approach for a restricted but critical aspect of run-time behavior: control/data flow due to interactions of the user with the application's GUI. A major portion of an Android application's logic is in code that processes GUI events. *GUI-related callback methods*, implemented by the application developer and invoked by the framework code, drive the overall control flow. Along this control flow, key categories of framework-managed objects (e.g., windows and widgets) flow back and forth between the framework code and the application code, as parameters of these callback methods. Aspects of this control/data flow are almost always a concern for existing static analyses for Android. For example, mismanagement of GUI control flow (e.g., the callbacks for an activity's lifecycle, which are discussed shortly) is one of the major sources of battery-drain defects [20, 25, 33].

One can draw an analogy with the classic problem of interprocedural control flow analysis in object-oriented programs: over twenty years of work have been dedicated to this problem, because of its foundational nature. We argue that control/data-flow analysis of *sequences of callbacks* in Android is similarly important because it serves as basis for many categories of analyses. In our work we focus on common kinds of GUI-related callbacks, but future investigations should consider additional aspects of this problem.

We propose to observe the run-time sequences of callbacks and their parameters, and to match them against static analysis abstractions. If a run-time sequence *cannot* be explained by a static analysis solution, it is clearly an example of unsoundness. Analyzing the number and characteristics of such examples provides valuable insights about the limitations of static analysis algorithms. The next section defines the details of the proposed matching and describes how it is applied to three existing static analyses: two versions of FlowDroid [3, 18] as well as our prior work on a window transition control-flow model in the GATOR analysis toolkit [30, 35]. Next, we present a study on six Android applications and discuss the observed unsoundness. We conclude with open questions and action items for program analysis researchers in this area.

2. Control Flow Unsoundness

2.1 Dynamic Trace

The execution of an Android application exhibits a variety of run-time events. A key aspect of this behavior is the *sequence of callbacks* from the framework code to the application code. An example of such a sequence is shown in Figure 1. The `@xyz` labels are identifiers of parameter objects and will be discussed later. This sequence was observed during the execution of a test case for the APV PDF reader [1]. In this test case, when the user starts the application, activity `ChooseFileActivity` is used to select a PDF file. An *activity* is a key building block of an Android application. Each activity corresponds to a window that interacts with the user. Callback methods `onCreate` and `onResume` defined in `ChooseFileActivity` are examples of *lifecycle callbacks* used the manage the activity's lifetime. Another example is `onCreateOptionsMenu`, which is a lifecycle callback for a menu window associated with the activity.

Another category of callback methods is *GUI event handlers*. For example, `onItemClick` is an event handler callback that is triggered when the user selects a list item from the list of PDF files displayed in the window of `ChooseFileActivity`. In this example the handler starts a new activity (`OpenFileActivity`) which displays the selected file in a new window. The last four callbacks

```
1  <ChooseFileActivity:
     void onCreate(android.os.Bundle)>@177466394
2  <ChooseFileActivity:
     void onResume()>@177466394
3  <ChooseFileActivity:
     boolean onCreateOptionsMenu(android.view.Menu)>@59596096
4  <ChooseFileActivity:
     void onItemClick(AdapterView,View,int,long)>@45225503
5  <OpenFileActivity:
     void onCreate(android.os.Bundle)>@59444588
6  <OpenFileActivity:
     void onResume()>@59444588
7  <OpenFileActivity:
     void onPause()>@59444588
8  <ChooseFileActivity:
     void onResume()>@177466394
9  <OpenFileActivity:
     void onStop()>@59444588
10 <OpenFileActivity:
     void onDestroy()>@59444588
```

Figure 1. Sample sequence of run-time callbacks.

in the example show what happens when the user presses the hardware BACK button to close the file and to return back to the list of files: an interleaving of lifecycle callbacks for `OpenFileActitivy` and `ChooseFileActivity` is triggered by the framework code, ending with an activity lifetime termination callback `onDestroy`.

Each callback method in this sequence completes execution before the next one is invoked. In other words, the lifetimes of these callback invocations are disjoint. In general, it is possible that during the invocation of a callback method c, an Android API call made by code in c (or by code in transitive callees of c) triggers a nested callback invocation of another callback method. In our current work we choose to focus only on the *top-level* callback invocations—that is, the ones that are not nested in other callback invocations. All callbacks shown in the example are top-level callbacks. Both our dynamic analysis and the static abstractions we consider model only these top-level invocations.

The sequence of top-level callback invocations can be easily obtained through simple instrumentation. Since we focus on the GUI-related control flow, in our implementation this instrumentation records only callbacks occurring in the GUI thread, which is the main application thread. A variety of callbacks can be observed in the resulting trace. It is natural to select certain *core features* and the run-time callbacks related only to those features. This would enable characterization of static analysis unsoundness with respect to these particular features. For our experiments, we choose the three core categories of callbacks described below. Of course, studies for other feature categories are essential for future work.

Category 1: lifecycle callbacks for activities. These methods are some of the most important and semantics-rich components of Android applications. We consider the standard activity lifetime callbacks `onCreate`, `onStart`, `onRestart`, `onResume`, `onPause`, `onStop`, and `onDestroy`.

Category 2: Category 1 + lifecycle callbacks for menus. Many windows in Android applications are menus [34] and they often trigger substantial changes to application state. In addition to Category 1, we also consider menu lifecycle callbacks `onCreateOptionsMenu`, `onPrepareOptionsMenu`, `onOptionsMenuClosed`, `onCreateContextMenu`, and `onContextMenuClosed`.

Category 3: Category 2 + GUI event handlers. In addition to lifecycle callbacks for activities and menus, the handlers of GUI events (e.g., clicking on a button or selecting a list item) are of considerable interest. A large number of static analyses investigate the effects of such event handlers. We focus on several Android interfaces that describe listeners for specific events (e.g., `OnClickListener`). Such interfaces define signatures for callback methods that processes particular GUI events on certain GUI widgets. For example, a callback method with signa-

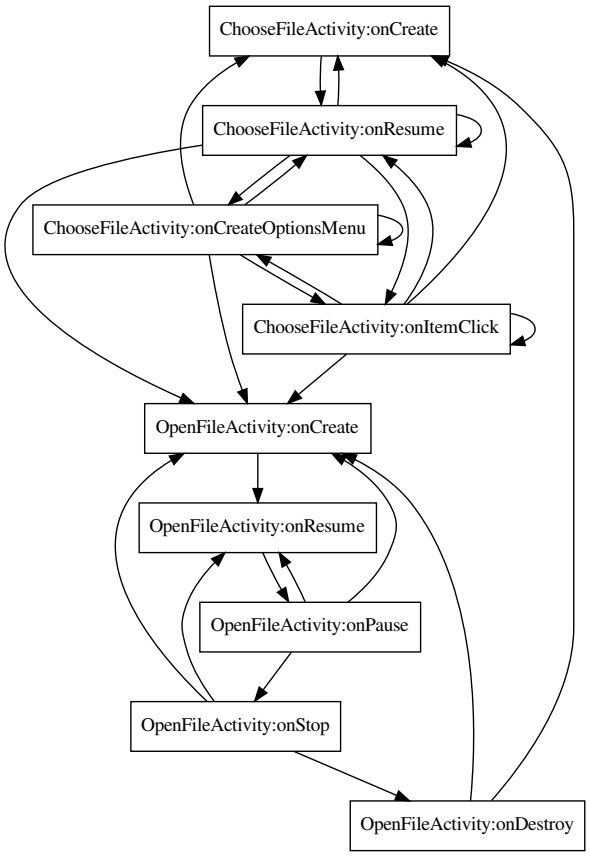

Figure 2. Callback sequence graph for FlowDroid.

ture `onClick(View)` is used to handle click events on the widget ("view" in Android terminology) provided as a parameter to the method. Another example is `onItemClick` from Figure 1, in which the second parameter is the list item that was clicked. We also consider two callback methods for clicking on items in menus: `onOptionsItemSelected` and `onContextItemSelected`.

Once a category has been chosen for investigation, the callback trace is filtered to contain only callbacks from this category. The resulting sequence $\mathcal{C} = c_1, c_2, \ldots, c_n$, where c_i is a callback method defined by the application, can be compared with a static solution. The measurement we have used in our study is the following: for a parameter p, each subsequence of $\mathcal{C}$ of the form $c_i, c_{i+1}, \ldots, c_{i+p}$ is compared against the static solution. The percentage of such subsequences that are *not* captured by this static solution is used to characterize the degree of unsoundness in the corresponding static analysis. The next section provides such measurements for six applications, three static analyses, and two values of p.

2.2 Static Analysis: FlowDroid and IccTA

FlowDroid [3] is a static taint analysis tool for Android applications. As part of the taint analysis, an artificial main method is created to simulate the callbacks triggered by the framework code. In our experiments we used the publicly available version 1.0 of FlowDroid[1] released in May 2013. We also considered the newer IccTA tool [18], which combines FlowDroid with an inter-component analysis. The analysis version was retrieved in February 2016.[2]

[1] github.com/secure-software-engineering/soot-infoflow-android/releases

[2] github.com/lilicoding/soot-infoflow-android-iccta

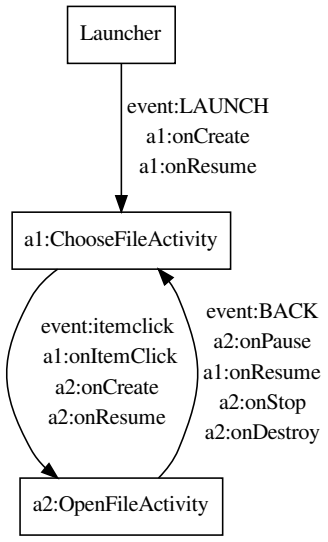

Figure 3. Window transition graph.

Both tools create the artificial main using the Jimple intermediate representation from the Soot framework [31]. Some of the statements in this method are invocations of lifecycle callback methods such as `onCreate` and GUI event handler callback methods such as `onItemClick`. The standard intraprocedural control-flow graph of this main method implicitly encodes sequences of callback invocations. To make these sequences explicit, we derive another control-flow representation, the *callback sequence graph* (CSG). In the CSG a node represents a callback method and an edge represents that, in some run-time execution, the target method may occur immediately after the source method.

To construct the CSG, we remove each CFG node that does *not* contain an invocation for a callback method from the category under investigation (i.e., Category 1, 2, or 3). When a node is removed, each predecessor is connected with each successor. The final graph (the CSG) directly represents possible callback sequences, as determined by this static analysis. Figure 2 shows the CSG for the running example, for Category 3 callbacks.

2.3 Static Analysis: GATOR

In recent work we introduced the window transition graph [35] as another control-flow representation for Android GUI control flow. This model has been used to perform static energy-drain defect detection [33], test generation for leaks [37], and responsiveness profiling [32]. The control-flow analysis is part of the publicly available GATOR toolkit for static analysis for Android.

Figure 3 shows this model for the running example. Graph nodes correspond to windows and edges correspond to transitions between windows. Each edge is annotated with the event that triggered the transition (e.g., "launch the application", "click on a list item", "press the hardware BACK button"). An edge is also labeled with the sequence of callbacks that occur due to this transition. An additional edge label (not discussed in this paper) captures the push/pop effects on the stack of currently-opened windows. This label can be used to identify valid paths [32, 33, 35, 37] in a manner similar to matching of calls and returns (i.e., call stack push/pop) in interprocedural analysis [28].

The callback sequence graph (CSG) is derived from this model. Each occurrence of a callback method along a transition is represented by a CSG node. A CSG edge connects (1) two adjacent callbacks along the same transition, and (2) the last callback of

one transition and the first callback of a successor transition. Next, methods *not* from the category under investigation (i.e., Category 1, 2, or 3) are removed, as was done earlier for FlowDroid's static model. In the example, the CSG for Category 3 is a chain containing 9 nodes, starting from `ChooseFileActivity:onCreate` and ending with `OpenFileActivity:onDestroy`. As with the CSG derived from the FlowDroid-created main method, this CSG captures all ordering constraints that are implied by GATOR's solution.

2.4 Matching with Dynamic Trace

Let $\mathcal{C} = c_1, c_2, \ldots, c_n$ be the dynamic trace of top-level callback invocations, filtered to contain only callbacks from a chosen category. Further, let $\mathcal{S}_p$ be the set of all contiguous subsequences of $\mathcal{C}$ of the form $c_i, c_{i+1}, \ldots, c_{i+p}$ where p is a parameter. The *coverage* of $\mathcal{S}_p$ is defined as $\kappa(\mathcal{S}_p) = |\mathcal{T}_p|/|\mathcal{S}_p|$ where $\mathcal{T}_p$ is the subset of $\mathcal{S}_p$ containing all and only callback sequences that are covered by the CSG. A run-time callback sequence $s = c_i, c_{i+1}, \ldots, c_{i+p}$ is covered by the static CSG if there exists a path in that graph whose node sequence matches s exactly.

3. Data Flow Unsoundness

Another significant complication is the data flow through parameters of callbacks. As illustrated in Figure 1, callback methods can have various parameters (including `this`). The values of these parameters are managed by the framework. Importantly, the same value could appear as a parameter of several callbacks. A trivial example are callbacks 1, 2, and 8 in Figure 1: all three have the same `this` parameter, which is an activity object managed "behind the scenes" by the framework. In this case it is easy to model statically the fact that all three parameters have the same value. However, in general, this data flow could involve a wide variety of objects, some of them carrying internal state that is read/written by multiple callbacks. A sound static analysis should model this data flow.

We consider one particular instance of this problem: for each of the callbacks described in the previous section, there is an object that represents the entity for which the callback implements a core functionality. For example, `this` for activity lifecycle methods refers to the activity object being processed. Similarly, the `Menu` parameter in `onCreateOptionsMenu` (event 3 in Figure 1) refers to the menu object whose lifetime is affected. In general, a lifecycle callback operates on a window that is easy to map to a callback parameter. Similarly, GUI event handlers from Category 3 process user actions on GUI widgets. For example, the `View` parameter in `onItemClick` (event 4 in Figure 1) is the list item being clicked.

Our instrumentation gathers the hash code of the corresponding callback parameter object (via `System.identityHashCode`), and stores it as part of the trace. In Figure 1, the hash codes of these activity/menu/widget objects are shown as labels `@xyz`. For all lifecycle callbacks and event handler callbacks in Category 3, we have manually created a specification of the parameter that should be recorded for each callback. This specification was created by examining the high-level semantics of the callbacks (based on Android documentation) and was used as input to the instrumentation component. When the instrumented application is executed, the resulting dynamic trace is of the form $\mathcal{C} = [c_1,o_1], [c_2,o_2], \ldots, [c_n,o_n]$ where o_k is the relevant run-time object for callback method c_k.

The three static analyses we evaluated create abstractions of these run-time objects. Each node in the CSG can now be considered a pair $[c,\hat{o}]$ where $\hat{o}$ is a static analysis entity which abstracts a dynamic object. For example, in FlowDroid and IccTA, `new X` expressions are created in the artificial main and they serve the role of $\hat{o}$. GATOR's window transition graph uses static abstractions for windows and widgets [29]. For all static analyses, we use the identity hash code of the corresponding analysis object to create CGS nodes of the form $[c,\hat{o}]$.

Given the generalized trace and CSG, consider a run-time subtrace of the form $s = [c_i,o_i], [c_{i+1},o_{i+1}], \ldots, [c_{i+p},o_{i+p}]$. Assume we have a CSG path $\hat{s} = [c_i,\hat{o}_i], [c_{i+1},\hat{o}_{i+1}], \ldots, [c_{i+p},\hat{o}_{i+p}]$ that matches the callback sequence $c_i, c_{i+1}, \ldots, c_{i+p}$ as defined earlier in Section 2.4. Does $\hat{s}$ match the objects o_k in s? To answer this question, we consider a basic feature of static analyses: there should exist an abstraction function α that maps each run-time entity to a corresponding static entity. We can attempt to construct a function that maps each o_k to the corresponding $\hat{o}_k$. If such a function α cannot be constructed, $\hat{s}$ does not match s.

More precisely, consider a partitioning of the elements of s into equivalence classes based on the run-time object being accessed. In the example in Figure 1, the equivalence classes are $\{1, 2, 8\}$ (i.e., all events that access activity `177466394`), $\{3\}$ for menu `59596096`, $\{4\}$ for widget `45225503`, and $\{5, 6, 7, 9, 10\}$ for activity `59444588`. If two elements $[c_k,o_k]$ and $[c_m,o_m]$ of run-time subtrace s belong to the same equivalence class, the corresponding elements $[c_k,\hat{o}_k]$ and $[c_m,\hat{o}_m]$ of path $\hat{s}$ must be such that $\hat{o}_k = \hat{o}_m$; in other words, we must have $\alpha(o_k) = \alpha(o_m)$.

4. Experimental Study

We studied six open-source Android applications for which we had prior experience creating tests with high GUI coverage. To obtain run-time traces, we manually developed tests to cover all windows and transitions between them. To ensure this coverage, we also examined the source code. The tests were executed on a Nexus 5X smartphone with Android 6.0. Instrumentation was inserted using Soot. The run-time trace contained method entry/exit events in the GUI thread together with identity hash codes of parameter objects. From this trace we derived the sequence of top-level callbacks.

4.1 Coverage Measurements

Table 1 shows measurements of coverage $\kappa_1 = \kappa(\mathcal{S}_1)$ and $\kappa_2 = \kappa(\mathcal{S}_2)$ of run-time callback sequences. That is, κ_1 shows how many run-time subsequences c_i, c_{i+1} were covered by the static solution, while κ_2 is the coverage for c_i, c_{i+1}, c_{i+2}. The first subtable corresponds to Category 1 (lifecycle callbacks for activities), the second one to Category 2 (added lifecycle callbacks for menus), and the third one to Category 3 (added GUI event handler callbacks). Table 2 shows similar measurements for coverage of run-time subsequences $[c_i,o_i], [c_{i+1},o_{i+1}]$ and $[c_i,o_i], [c_{i+1},o_{i+1}], [c_{i+2},o_{i+2}]$.

In all cases, the run-time sequences correspond to relatively simple aspects of run-time behavior. For example, κ_1 in Table 1 corresponds to pairs of consecutive callbacks observed at run time. Even for this simple run-time behavior, the unsoundness of the static analyses is quite clear. It is important to note that FlowDroid and IccTA were designed for the purpose of taint analysis, and the unsoundness they exhibit may be harmless in this context. However, they may not be well suited for other purposes that require more general control/data flow modeling. GATOR also shows various limitations, as discussed below.

4.2 Causes of Unsoundness

We manually examined some of the cases where coverage was less than 100%. While this is not a comprehensive study, it provides some insights about the limitations of these analyses.

GATOR *Unsoundness* One of the reasons for low coverage in `BarcodeScanner` is that the control-flow analysis [35] does not consider implicit intents. Intents are the standard mechanism for starting a new activity. A explicit intent contains the name of the activity being started; GATOR analyzes them and represents their effects. Implicit intents "declare a general action to perform, which allows a component from another app to handle it" [12]. However, it is possible that a component from the same app is used

App	GATOR		FlowDroid		IccTA	
	κ_1	κ_2	κ_1	κ_2	κ_1	κ_2
APV	1.00	1.00	0.41	0.14	0.35	0.10
BarcodeScanner	0.86	0.74	0.64	0.41	0.64	0.41
OpenManager	1.00	1.00	1.00	1.00	1.00	1.00
SuperGenPass	1.00	1.00	0.83	0.67	0.83	0.67
TippyTipper	1.00	1.00	0.41	0.09	0.41	0.09
VuDroid	1.00	1.00	0.50	0.00	0.50	0.00
APV	0.92	0.77	0.54	0.39	0.50	0.35
BarcodeScanner	0.88	0.77	0.69	0.47	0.54	0.30
OpenManager	1.00	1.00	0.89	0.75	0.89	0.75
SuperGenPass	1.00	1.00	0.90	0.82	0.90	0.82
TippyTipper	1.00	1.00	0.41	0.11	0.41	0.11
VuDroid	1.00	1.00	0.40	0.00	0.40	0.00
APV	0.90	0.71	0.45	0.32	0.59	0.45
BarcodeScanner	0.85	0.75	0.67	0.50	0.61	0.43
OpenManager	0.95	0.92	0.50	0.33	0.77	0.58
SuperGenPass	1.00	1.00	0.82	0.75	0.82	0.75
TippyTipper	1.00	1.00	0.39	0.11	0.39	0.11
VuDroid	1.00	1.00	0.56	0.29	0.22	0.00

Table 1. Coverage of callback sequences.

App	GATOR		FlowDroid		IccTA	
	κ_1	κ_2	κ_1	κ_2	κ_1	κ_2
APV	0.97	0.93	0.55	0.20	0.55	0.20
BarcodeScanner	0.88	0.73	0.62	0.39	0.62	0.39
OpenManager	1.00	1.00	1.00	1.00	1.00	1.00
SuperGenPass	1.00	1.00	0.79	0.69	0.79	0.69
TippyTipper	1.00	1.00	0.46	0.11	0.46	0.11
VuDroid	1.00	1.00	0.43	0.00	0.43	0.00
APV	0.81	0.53	0.69	0.47	0.69	0.47
BarcodeScanner	0.90	0.76	0.67	0.46	0.54	0.30
OpenManager	1.00	0.40	0.35	0.25	0.35	0.25
SuperGenPass	1.00	0.86	0.85	0.77	0.85	0.77
TippyTipper	1.00	1.00	0.45	0.12	0.45	0.12
VuDroid	1.00	1.00	0.33	0.00	0.33	0.00
APV	0.84	0.56	0.68	0.49	0.72	0.54
BarcodeScanner	0.87	0.79	0.66	0.50	0.62	0.44
OpenManager	0.98	0.73	0.33	0.14	0.53	0.19
SuperGenPass	1.00	0.87	0.81	0.74	0.81	0.74
TippyTipper	1.00	1.00	0.43	0.12	0.43	0.12
VuDroid	0.93	0.92	0.40	0.08	0.20	0.00

Table 2. Coverage of callback sequences and parameter flow.

to handle the intent. In `BarcodeScanner`, a GUI event handler uses an implicit intent to start a new activity from this application. The corresponding transition is missing from the window transition graph. Such unsoundness could be eliminated by employing one of the many existing intent analyses for Android (e.g., [24]).

Another source of unsoundness is the use of `ActionBar`. This feature, introduced in Android 3.0, changes the behavior of an options menu (a menu associated with an activity). Lifecycle callback `onCreateOptionsMenu` is invoked when the menu is initialized. In earlier Android versions, this happens when the user presses the hardware MENU button. However, with the introduction of action bars, the menu may be shown when the activity itself is shown, without any action from the user. Whether this behavior occurs depends on screen size and application layout. The control-flow analysis in GATOR assumes the menu is only shown when clicking the MENU button, and does not capture the possibility that `onCreateOptionsMenu` happens directly after activity creation. This example illustrates the challenges due to Android evolution.

FlowDroid and IccTA Unsoundness In both approaches, the interleaving of callbacks across two activities is not represented in its most general form. Consider events 7, 8, and 9 from Figure 1. These callbacks occur when the hardware BACK button is pressed to close `OpenFileActivity` and return to `ChooseFileActivity`. Events 7 and 9 are callbacks on the first activity while event 8 is a callback on the second one. The artificial main method soundly encodes callback ordering constraints inside each individual activity, but not the interleaving 7, 8, 9. Some client analyses may be unaffected by this source of unsoundness, but this conceptual limitation should be taken into account by designers of new clients.

In addition to this issue, we discovered that FlowDroid's main method misses several `onClick` event handlers, as well as an instance of `onItemSelected` handler. Further, one lifecycle callback `onCreateOptionsMenu` is missed because it is defined in an activity class (`BaseViewerActivity` in `VuDroid`) that is not declared in the XML manifest file and thus not analyzed. However, this class is the superclass of two other activities in the code. Similar omissions were observed in IccTA's main method.

Discussion This study illustrates the difficulty of achieving sound static analysis for Android GUI control/data flow. Of course, the scope of the study is limited with respect to observed run-time behaviors, analyzed features, and metrics of unsoundness. Further, this style of evaluation cannot be used to argue that one analysis is "better" than another one without considering the expected clients and the overall precision (i.e., what parts of the static solution are feasible at run time).[3] Nevertheless, these results provide a motivation for more studies and new developments in static analysis for Android. Section 6 outlines thoughts for some such developments.

5. Related Work

Our discussion focuses on modeling of GUI control flow (and sometimes the related data flow) in prior Android analyses. We do not aim to present a comprehensive description and comparison of existing control-flow models. Such a study, which should focus on both theoretical properties as well as experimental comparisons, is a highly-desirable target for future work.

Many security analyses [7, 9, 10, 13, 15, 22, 23] capture some aspects of Android control/data flow (e.g., possible sequences of callbacks) but do not provide a comprehensive model, nor do they make arguments about soundness. Other static analyses also model the sequences of callbacks in Android, for the purposes of GUI exploration (e.g., [4, 36]), responsiveness (e.g., [19, 32]), leak analysis (e.g., [14]), and static checking (e.g., [26, 38]); all these approaches employ *ad hoc* control-flow modeling that lacks generality. Attempts to formally capture aspects of Android operational semantics (e.g., [27, 29]) have limited scope and much work remains to be done in this direction. Recent work on automatic creation of semantic models for Android framework code [2, 5, 17] presents a promising step in this direction.

As discussed earlier, FlowDroid [3] and IccTA [18] use an artificial main method to represent callback sequences. This approach does not model the full generality of event handlers or the interleaving of callbacks from multiple activities. Our earlier work on the window transition graph in GATOR [34, 35] presents another approach for modeling the GUI control flow, but it also suffers from unsoundness problems, as described in Section 4.

Studies of unsoundness in static analysis are highly desirable. A recent example is work by Christakis et al. [8] in which unsound assumptions in the Clousot static analyzer for .NET are examined at run time for violations. The evaluation of the Droidel tool [5] compares a dynamic call graph with the static call graphs from this tool and from FlowDroid. There is also a large body of work on improving soundness for challenging features such as reflection, including techniques that leverage dynamic information (e.g., [6]).

[3] In manual studies [35], this precision for GATOR appears to be high.

6. Conclusions and Future Work

This is a very preliminary study, but it does highlight the challenges for static analysis researchers in the increasingly important area of Android software. Addressing these challenges can lead to several interesting directions for future work. One natural first step is to establish a suite of microbenchmarks. For each Android feature from a well-chosen set of core features, there should be microbenchmarks containing (1) code that uses this feature, (2) test cases that execute the feature under several usage scenarios, and (3) run-time traces for several Android API versions. Popular core features could be selected by mining textual documentation (e.g., from `developer.android.com`) and a large corpus of apps.

The microbenchmarks, as well as real-world apps, can then be used to understand the (un)soundness of various static analyses. This would require additional metrics of coverage, some of which should embed client-specific notions of soundness (e.g., "sound for solving problem X"). There is a significant scope for performing new studies similar in spirit to the one presented here, but with more refined metrics and deeper experiments.

One significant challenge is the lack of formal semantics for Android run-time behavior. Some initial attempts for semantics definitions have been made [27, 29] but much more work is needed in terms of (1) covered Android features, and (2) validating the semantics against a rich variety of run-time behaviors. Finally, it is important to be able to identify and eliminate new aspects of unsoundness for an existing analysis, caused by the evolution of the Android framework. This could be done through incremental (and, ideally, automated) soundness checking and patching.

Acknowledgments We thank the SOAP reviewers for their valuable feedback. This material is based upon work supported by the U.S. National Science Foundation under CCF-1319695 and CCF-1526459, and by a Google Faculty Research Award.

References

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

[2] S. Arzt and E. Bodden. StubDroid: Automatic inference of precise data-flow summaries for the Android framework. In *ICSE*, 2016.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, 2014.

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

[5] S. Blackshear, A. Gendreau, and B.-Y. E. Chang. Droidel: A general approach to Android framework modeling. In *SOAP*, 2015.

[6] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

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

[8] M. Christakis, P. Müller, and V. Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *VMCAI*, 2015.

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

[10] 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.

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

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

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

[14] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and detecting resource leaks in Android applications. In *ASE*, 2013.

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

[16] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 23(3), May 2001.

[17] J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama. Synthesizing framework models for symbolic execution. In *ICSE*, 2016.

[18] L. Li, A. Bartel, T. F. Bissyande, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *ICSE*, 2015.

[19] Y. Lin, C. Radoi, and D. Dig. Retrofitting concurrency for Android applications through refactoring. In *FSE*, 2014.

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

[21] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 58(2), Jan. 2015.

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

[23] 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.

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

[25] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys*, 2012.

[26] E. Payet and F. Spoto. Static analysis of Android programs. *IST*, 54 (11), 2012.

[27] E. Payet and F. Spoto. An operational semantics for Android activities. In *PEPM*, 2014.

[28] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12), 1998.

[29] A. Rountev and D. Yan. Static reference analysis for GUI objects in Android software. In *CGO*, 2014.

[30] A. Rountev, D. Yan, S. Yang, H. Wu, Y. Wang, and H. Zhang. GATOR: Program analysis toolkit for Android. `web.cse.ohio-state.edu/presto/software`.

[31] 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 *CC*, 2000.

[32] Y. Wang and A. Rountev. Profiling the responsiveness of Android applications via automated resource amplification. In *MobileSoft*, 2016.

[33] H. Wu, S. Yang, and A. Rountev. Static detection of energy defect patterns in Android applications. In *CC*, 2016.

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

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

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

[37] H. Zhang, H. Wu, and A. Rountev. Automated test generation for detection of leaks in Android applications. In *AST*, 2016.

[38] S. Zhang, H. Lü, and M. D. Ernst. Finding errors in multithreaded GUI applications. In *ISSTA*, 2012.