Static Detection of Energy Defect Patterns in Android Applications

Haowei Wu Shengqian Yang Atanas Rountev
Ohio State University, USA
{wuhaow,yangs,rountev}@cse.ohio-state.edu

Abstract

For static analysis researchers, Android software presents a wide variety of interesting challenges. The target of our work is static detection of energy-drain defects in Android applications. The management of energy-intensive resources (e.g., GPS) creates various opportunities for software defects.

Our goal is to detect statically "missing deactivation" energy-drain defects in the user interface of the application. First, we define precisely two patterns of run-time energy-drain behaviors, based on modeling of Android GUI control-flow paths and energy-related listener leaks along such paths. Next, we define a static detection algorithm targeting these patterns. The analysis considers valid interprocedural control-flow paths in a callback method and its transitive callees, in order to detect operations that add or remove listeners. Sequences of callbacks are then analyzed for possible listener leaks. Our evaluation considers the detection of GUI-related energy-drain defects reported in prior work, as well as new defects not discovered by prior approaches. In summary, the detection is very effective and precise, suggesting that the proposed analysis is suitable for practical use in static checking tools for Android.

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

General Terms Algorithms, experimentation, measurement

Keywords Android, GUI analysis, static analysis, energy

1. Introduction

The computing field has changed significantly in the last few years due to the exponential growth in the number of mobile devices such as smartphones and tablets. In this space, Android is the dominant platform [12]. For static analysis researchers, Android software presents a wide variety of interesting challenges, both in terms of foundational control-flow and data-flow analysis techniques and in terms of specific analyses targeting software correctness, robustness, performance, and security.

The target of our work is *static detection of energy-drain defects in Android applications*. For mobile devices, the management of energy-intensive resources (e.g., GPS) burdens the developer with "power-encumbered programming" [36] and creates various opportunities for software defects. Static detection of such defects is of significant value. Common battery-drain defects—"no-sleep" [36] and "missing deactivation" [4, 27]—are due to executions along which *an energy-draining resource is activated but not properly deactivated*. Such dynamic behaviors can be naturally stated as properties of control-flow paths, and thus present desirable targets for static control-flow and data-flow analyses.

State of the art Despite the clear importance of energy-drain defects, there is limited understanding of how to detect such defects with the help of program analysis. Early work on this topic [36] defines a data-flow analysis to identify relevant API calls that turn on some energy-draining resource, and to search for *no-sleep code paths* along which corresponding turn-off/release calls are missing. For this approach the control-flow analysis of possible execution paths is of critical importance. However, due to the GUI-based and event-driven nature of Android applications and the complex interactions between application code and framework code through sequences of callbacks, static control-flow modeling is a very challenging problem. This prior work employs an ad hoc control-flow analysis that exhibits significant lack of generality and precision, and involves manual effort by the user. An alternative is to use dynamic analyses to detect certain categories of energy-drain defects [4, 27]. Such defects also correspond to execution paths in which a sensor (e.g., the GPS) is not put to sleep appropriately, often because of mismanagement of complex callbacks from the Android framework to the application code. However, such run-time detection critically depends on the ability to trigger the problematic behavior. This requires comprehensive GUI run-time exploration, which is a very challenging problem for an automated analysis [8]. It is highly desirable to develop static analyses that employ general control-flow modeling of all possible run-time behaviors, together with detection of the energy-drain patterns along such behaviors.

Our proposal We aim to develop a general static analysis approach for detecting certain common categories of energy-drain defects. Specifically, we aim to detect "missing deactivation" behaviors in the user interface thread of the application. This is the main thread of the application and the majority of application logic is executed in it. While such problematic behaviors may also occur in threads running concurrently with the UI thread, the current state of the art for control-flow analysis of multithreaded Android control flow is still very underdeveloped and there is no conceptual clarity on how Android-specific asynchronous constructs (e.g., asynchronous tasks and long-running services) should be modeled statically. Thus, in our current work we focus only on the behavior of the UI thread and the callbacks executed in that thread. Due to recent advances in static control-flow analysis of Android UI behavior [48], the development of static detectors for such energy-drain defects is feasible for the first time. Future work on more

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

CC'16, March 17–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-4241-4/16/03...
http://dx.doi.org/10.1145/2892208.2892218

general static control-flow analysis for multithreaded Android executions will also enable generalizations of our current techniques to additional categories of energy-drain defects.

The proposed approach is based on three key contributions. First, we define precisely two patterns of run-time energy-drain behaviors (Section 2). The definition is based on formal definitions of relevant aspects of Android GUI run-time control flow, including modeling of GUI events, event handlers, transitions between windows, and the associated sequences of callbacks. This modeling allows us to define the notion of a leaking control-flow path and two defect patterns based on it. These patterns are related to the (mis)use of Android location awareness capabilities (e.g., GPS location information). Location awareness is a major contributor to energy drain [16] and in prior work on dynamic defect detection [27] has been identified as the predominant cause of energy-related defects in UI behavior. Our definition of defect patterns is motivated by case studies from this prior work and by our own analysis of these case studies. However, our careful formulation of these patterns is new and provides a valuable contribution to the state of the art. Furthermore, our control-flow modeling is significantly more general than any prior technique.

The second contribution of our approach is a static defect detection algorithm (Section 3). As a starting point, we use the window transition graph (WTG), a static GUI control-flow model we proposed in prior work [48]. Based on this model, the analysis considers valid interprocedural control-flow paths in each callback method and its transitive callees, in order to detect operations that add or remove location listeners. Sequences of window transitions and their callbacks are then analyzed for possible listener leaking behaviors based on the two patterns mentioned earlier.

The final contribution of our work is a study of the effectiveness of the proposed static detection (Section 4). We aim to determine how well the analysis discovers GUI-related energy-drain defects reported in prior work, as well as new defects not discovered by prior approaches. Our evaluation on 15 Android applications indicates that the static detection is very effective and is superior to dynamic detection. Furthermore, all but one of the reported problems are real defects. The evaluation also shows that the cost of the analysis is low. This high precision and low cost suggest that the proposed approach is suitable for practical use in static checking tools for Android.

2. Energy Defect Patterns in Android Software

2.1 Relevant Features of Android GUIs

This section describes the features of Android run-time semantics that are relevant to the defects considered in this work. Figure 1 summarizes our notation for these features. We focus on the event-driven control flow in the GUI of the application (i.e., in the main application thread), since our work targets defects that occur along this control flow. Figure 2 and Figure 3 contain two code examples to illustrate these features.

Windows and views *Activities* are core components of Android applications, defined by subclasses of `android.app.Activity`. An activity displays a window containing several GUI widgets. A widget (a "view" in Android terminology) is an object from a subclass of `android.view.View`.

▷ *Example:* Figure 2 shows an example derived from an energy-drain defect we found in the DroidAR application analyzed in prior work on the GreenDroid dynamic defect detection tool [27]. This particular defect is new, and was not reported in that prior work. The defect is due to the lack of a listener-remove operation at the end of the lifetime of activity `DemoLauncher`. When button `btnRun` is clicked, the `onClick` handler registers a location listener (at line 16) but this listener is not removed even after the activity is destroyed. As a result, location data (e.g., GPS reads) is sensed even after it is not needed anymore, which will drain the battery.

w	$\in \mathbf{Win}$	window
v	$\in \mathbf{View}$	view
$e = [v,k]$	$\in \mathbf{Event}$	widget event on v
$e = [w,k]$	$\in \mathbf{Event}$	default event on w
c	$\in \mathbf{Cb}$	callback method
$[c,o]$	$\in \mathbf{Cb} \times (\mathbf{Win} \cup \mathbf{View})$	callback invocation
s	$\in \mathbf{Cbs}$	callback inv. sequence
$t = [w,w']$	$\in \mathbf{Trans}$	window transition
$\epsilon(t)$	$\in \mathbf{Event}$	event that triggered t
$\sigma(t)$	$\in \mathbf{Cbs}$	callback sequence for t
$\delta(t)$	$\in (\{push, pop\} \times \mathbf{Win})^*$	window stack changes
T	$\in \mathbf{Trans}^+$	transition sequence
l	$\in \mathbf{Lst}$	listener
$op = [a,r]$	$\in \mathbf{Op}$	add/remove listener APIs
A	$\subseteq \mathbf{Op}_a \times \mathbf{Lst}$	added listeners
R	$\subseteq \mathbf{Op}_r \times \mathbf{Lst}$	removed listeners

Figure 1: Notation for run-time semantics.

Class `DemoLauncher` is an example of an activity. In this case this is the start activity of the application: it is started by the Android launcher when the user launches the application. The `onResume` lifecycle callback (discussed shortly) retrieves a button widget `btnRun` at line 3. This widget is then associated with an event handler callback `onClick`, defined in an anonymous class that implements interface `OnClickListener` (lines 4–5). Parameter `v` in this callback refers to the button widget. ◁

We also consider the two other common categories of Android windows: menus and dialogs. Instances of menu classes represent short-lived windows associated with activities ("options" menus) and widgets ("context" menus). An example presented later in the paper illustrates the use of menus. A dialog is an object from some subclass of `android.app.Dialog`. Both menus and dialogs require users to take an action before they can proceed [15]. A menu/dialog implements a simple interaction with the user, and its lifetime is shorter than activity lifetime. The last activity that was displayed before a menu/dialog was displayed is the *owner activity* of this menu/dialog. The lifetime of a menu or a dialog is contained within the lifetime of its owner activity.

We will use $\mathbf{Win}$ to denote the set of all run-time windows (activities, menus, and dialogs) and $\mathbf{View}$ for the set of all run-time widgets in these windows (Figure 1).

Events Each $w \in \mathbf{Win}$ can respond to several events. *Widget events* are of the form $e = [v,k]$ where $v \in \mathbf{View}$ is a widget and k is an event kind. For the example in Figure 2 we have event $[br,click]$ where br is the `Button` instance referenced by `btnRun`.

We also consider five kinds of *default events*. Event $back$ corresponds to pressing the hardware BACK button, which closes the current window w and typically (but not always) returns to the window that opened w.[1] Event $rotate$ shows that the user rotates the screen, which triggers various GUI changes. Event $home$ abstracts a scenario there the user switches to another application and then resumes the current application (e.g., by pressing the hardware HOME button to switch to the Android application launcher, and then eventually returning back to the application). Event $power$ represents a scenario where the device screen is turned off by pressing the hardware POWER button, followed by device reactivation. Event $menu$ shows the pressing of the hardware MENU button to

[1] In some scenarios (e.g., callback `onBackPressed` is defined) the window is not closed. We have not observed such scenarios in the analyzed apps.

```
1  class DemoLauncher extends Activity {
2    void onResume() {
3      Button btnRun = ...;
4      btnRun.setOnClickListener(new OnClickListener() {
5        void onClick(View v) { run(); } }); }
6    // === Other possible lifecycle callbacks: onCreate,
7    // === onDestroy, onStart, onRestart, onPause
8    void run() {
9      EventManager.getInstance().registerListeners(); } }
10
11 class EventManager implements LocationListener {
12   static EventManager instance = new EventManager();
13   static getInstance() { return instance; }
14   void registerListeners() {
15     LocationManager lm = ...;
16     lm.requestLocationUpdates(this); } }
```

Figure 2: Example derived from the DroidAR application.

display an options menu. A default event will be represented as $e = [w,t] \in \mathbf{Win} \times \{back, rotate, home, power, menu\}$ where w is the currently-active window. We will use $\mathbf{Event}$ to denote the set of all widget events and default events. In Figure 2 we have five default events [DemoLauncher,. . .], but since the activity does not define an options menu, event $menu$ does not have any effect.

Callbacks Each $e \in \mathbf{Event}$ triggers a sequence of *callback invocations* that can be abstracted as $[c_1,o_1][c_2,o_2]\ldots[c_m,o_m]$. Here c_i is a callback method defined by the application, and o_i is a run-time object on which c_i was triggered. Note that each of these invocations completes before the next one starts—that is, their lifetimes are not nested within each other, but rather they are disjoint. The actual invocations are performed by event-processing logic implemented inside the Android framework code.

We consider two categories of callbacks. *Widget event handler callbacks* respond to widget events; an example is `onClick` in Figure 2. *Lifecycle callbacks* are used for lifetime management of windows. For example, creation callback `onCreate` indicates the start of the activity's lifetime, and termination callback `onDestroy` indicates end of lifetime. Menus and dialogs can also have create/terminate callbacks.

▷ *Example:* In Figure 2 event $[br,click]$ (br is the `Button` object referenced by `btnRun`) will cause a widget event handler callback invocation [onClick,br]. In this example the callback sequence contains only this invocation. However, for the sake of the example, suppose that `onClick` invoked an Android API call to start some new activity a. Also, for illustration, suppose that the source activity `DemoLauncher` and the target activity a both define the full range of activity lifecycle callbacks (listed for completeness at lines 6–7 in Figure 2). Then the callback invocation sequence would be [onClick,br][onPause,DemoLauncher] [onCreate,a][onStart,a][onResume,a][onStop,DemoLauncher]; this sequence can be observed via `android.os.Debug` tracing.

If after $[br,click]$ the next event was $[a,back]$—that is, the BACK button was pressed to close a and return to `DemoLauncher`—the sequence would be [onPause,a] [onRestart,DemoLauncher] [onStart,DemoLauncher][onResume,DemoLauncher][onStop,a] [onDestroy,a]. As seen from these examples, there can be a non-trivial sequence of callback invocations in response to a single GUI event. ◁

Window transitions We use the term *run-time window transition* to denote a pair $t = [w,w'] \in \mathbf{Win} \times \mathbf{Win}$ showing that when window w was active and interacting with the user, a GUI event occurred that caused the new active window to be w' (w' may be the same as w). Each transition t is associated with the event $\epsilon(t) \in \mathbf{Event}$ that caused the transition and with $\sigma(t)$, a sequence of callback invocations $[c_i,o_i]$.

There are two categories of callback invocation sequences for Android GUI transitions. The first case is when event $\epsilon(t)$ is a widget event $[v,k]$ where v is a widget in the currently-active window w. In this case $\sigma(t)$ starts with $[c_1,v]$ where c_1 is the callback responsible for handling events of type k on v. The rest of the sequence contains $[c_i,w_i]$ with c_i being a lifecycle callback on some window w_i. In general, the windows w_i whose lifecycles are affected include the source window w, the target window w', as well as other related windows (e.g., the owner activity of w). In the running example, a self-transition t for `DemoLauncher` is triggered by event $[br,click]$, resulting in $\sigma(t) = $ [onClick,br]. Following the hypothetical example from above, if `onClick` opens another activity a, the transition would be from `DemoLauncher` to a, with $\sigma(t)$ as listed above: [onClick,br]. . . [onStop,DemoLauncher].

The second category of callback sequences is when $\epsilon(t)$ is a default event $[w,k]$ on the current window w. In this case all elements of $\sigma(t)$ involve lifecycle callbacks. For example, event $home$ on `DemoLauncher` triggers a self-transition t with $\sigma(t)$ containing invocations of `onPause`, `onStop`, `onRestart`, `onStart`, `onResume` on that activity. Additional details of the structure of these callback sequences are presented in our earlier work [46, 48].

Window stack Each transition t may open new windows and/or close existing ones. This behavior can be modeled with a *window stack*: the stack of currently-active windows.[2] Each transition t can modify the stack by performing window push/pop sequences. These effects will be denoted by $\delta(t) \in (\{push, pop\} \times \mathbf{Win})^*$. In the examples presented in this paper, the effects of a transition t are relatively simple: for example, opening a new window w represented by $push\ w$, or closing the current window w represented by $pop\ w$. In the simplest case, as in the self-transition t from Figure 2, $\delta(t)$ is empty. However, our prior work [48] shows that in general these effects are more complex: $\delta(t)$ could be a (possibly empty) sequence of window pop operations, followed by an optional push operation. These operations could involve several windows and can trigger complicated callback sequences.

Transition sequences Consider any sequence of transitions $T = \langle t_1, t_2, \ldots, t_n \rangle$ such that the target of t_i is the same as the source of t_{i+1}. Let $\sigma(T)$ be the concatenation of callback sequences $\sigma(t_i)$; similarly, let $\delta(T)$ be the concatenation of window stack update sequences $\delta(t_i)$. Sequence T is *valid* if $\delta(T)$ is a string in a standard context-free language [39] defined by

$$Valid \rightarrow Balanced\ Valid \mid push\ w_i\ Valid \mid \epsilon$$

where $Balanced$ describes balanced sequences of matching push and pop operations

$$Balanced \rightarrow Balanced\ Balanced \mid push\ w_i\ Balanced\ pop\ w_i \mid \epsilon$$

2.2 Adding and Removing of Listeners

The callbacks invoked during window transitions can perform a variety of actions. Our work considers actions that may affect energy consumption. In particular, we focus on *add-listener* and *remove-listener* operations related to location awareness. Such actions have been considered by GreenDroid [27], an existing dynamic analysis tool for detection of energy defects in Android applications. Since almost all GUI-related energy-drain defects reported in this prior work are due to location awareness, focusing on such defects allows us to perform direct comparison with the results from this existing study.

Relevant Android APIs The standard mechanism for obtaining information about the location of the user (e.g., using GPS data)

[2] Features such as launch modes for activities [13] can lead non-LIFO behaviors, but they do not appear to be commonly used [44].

is by registering a *location listener* with the framework's location manager. The listener implements callback methods that are invoked when relevant changes happen. Registration is done through API calls such as `requestLocationUpdates`, with the location listener provided as a parameter. The listener can be removed by calling `removeUpdates` and using the listener as a parameter. A number of other Android APIs have similar effects. For example, when an application is displaying a map, it could display an overlay of the current user location on the map by calling `enableMyLocation` on an overlay object, in which case this object becomes a listener for location updates. A subsequent call to `disableMyLocation` stops this listening.

▷ *Example:* In Figure 2 callback [`onClick`,br] invokes `run`, which in turn invokes `registerListeners` on an instance of `EventManager`. This instance (created at line 12) is a listener object that is registered for location updates at line 16. However, this listener is never removed by any other code in the activity. In particular, if the user exits `DemoLauncher`—e.g., by pressing the hardware BACK button to exit the application—the listener will remain registered and will drain the battery. We have confirmed this incorrect behavior through testing. ◁

The standard guidelines for building location-aware applications warn the developers to *"always beware that listening for a long time consumes a lot of battery power"* [16]. Our goal is to model the addition and removal of location listeners along sequences of window transitions (and their related callbacks), in order to identify problematic behaviors that may lead to extended periods of location listening. We formalize the relevant run-time features as follows (also see Figure 1). Let $\mathbf{Lst}$ be the set of all run-time objects l that are location listeners. Let $\mathbf{Op}$ be the set of pairs $[a,r]$ where a is an API method for adding a listener and r is the corresponding API method for removing that listener. We will use $\mathbf{Op}_a$ to denote $\{a \mid [a,r] \in \mathbf{Op}\}$; $\mathbf{Op}_r$ is defined similarly. For example, [`requestLocationUpdates`,`removeUpdates`] $\in \mathbf{Op}$.

Leaking sequences Consider $s = [c_1,o_1][c_2,o_2]\ldots[c_m,o_m]$, a callback sequence observed during some window transitions. Recall that c_i is a callback method and o_i is a view/window on which c_i is called. Let A_i be the set of pairs $[a,l] \in \mathbf{Op}_a \times \mathbf{Lst}$ such that add-listener method a was invoked on listener l during the execution of c_i on o_i, and the rest of the execution of c_i did not invoke r on l for any $[a,r] \in \mathbf{Op}$. In other words, c_i (or its transitive callees) invoked a and provided l as a parameter, and subsequently did not invoke on l any remove-listener method r that matches a. One can draw an analogy with the standard compiler notion [1] of a downward-exposed definition in a basic block (i.e., a definition that reaches the exit of the block). Similarly, we will use *downward-exposed* to denote any $[a,l]$ that was not killed by a subsequent $[r,l]$ in c_i and its transitive callees, and thus reached the exit of c_i.

In the running example, for the callback sequence containing only [`onClick`,br], the corresponding set A_1 contains one element: [`requestLocationUpdates`,l] where l is created at line 12.

Similarly, for callback invocation $[c_i,o_i]$, let R_i contain $[r,l] \in \mathbf{Op}_r \times \mathbf{Lst}$ such that remove-listener method r was invoked on l during the callback execution. Note that the definition of R_i could have included the following additional condition: "in the execution of c_i (and its transitive callees) $[r,l]$ was not preceded by a matching $[a,l]$". Such a condition would have made the definition of R_i similar to the definition of A_i. However, such a condition is not necessary because in Android it is possible for a single $[r,l]$ to be preceded by several matching $[a,l]$, occurring over multiple callbacks, including the callback c_i that invokes $[r,l]$. That single remove operation "cancels" all preceding add operations. Thus, it is irrelevant whether c_i contains a preceding $[a,l]$.

DEFINITION 1. *Given a callback invocation sequence s of length m, let $A_1, A_2, \ldots, A_m$ be its sequence of add-listener sets and $R_1, R_2, \ldots, R_m$ be its sequence of remove-listener sets. Sequence s leaks listener l if there exists an add-listener operation $[a,l] \in A_i$ such that for each $j > i$ there does not exist a matching $[r,l] \in R_j$.*

Leaking callback sequences are typically harmless: they represent legitimate needs to receive updated information about the location of the device. For example, in Figure 2, the click event on the button causes a window self-transition with a leaking callback sequence (containing only [`onClick`,br]), but of course this is the intended behavior. However, not all leaking sequences are desired. We define two patterns of leaking sequences that represent potential defects. These patterns are motivated by case studies from the work on GreenDroid, and by our own analysis of these case studies.

2.3 Pattern 1: Lifetime Containment

The informal definition of this pattern is as follows: if an activity adds a listener, the listener should be removed before that activity is destroyed. A similar pattern has been defined informally as part of the GreenDroid tool, but our goal here is to state it precisely in order to allow the development of static detection algorithms. Note that since menus and dialogs are intended to be short-lived, their lifetimes cannot be expected to contain the lifetime of listener registration; thus, the pattern is defined only for activities.

Consider a window transition sequence $T = \langle t_1, t_2, \ldots, t_n \rangle$ and recall that $\delta(T)$ is the concatenation of window stack push/pop operations $\delta(t_i)$. Sequence T represents a lifetime of an activity w if $\delta(t_1)$ contains an operation $push\ w$, $\delta(t_n)$ contains an operation $pop\ w$, and the sequence of push/pop operations between these two operations in $\delta(T)$ is balanced (as defined by non-terminal $Balanced$ described earlier).

DEFINITION 2. *Suppose T represents a lifetime of an activity w. Consider the callback invocation sequence $\sigma(T)$ and its subsequence $s = [c_1,w]\ldots[c_m,w]$ where c_1 is the creation callback for w and c_m is the termination callback for w. If s is leaking a listener l, and the corresponding add-listener operation $[a,l]$ occurred when the top-most activity on the window stack was w, then T matches Pattern 1.*

Note that this definition does not say "the top element on the window stack was w". Here we allow for some menus/dialogs (owned by w) to be on top of w in the window stack at the time when the listener is added. Since menus and dialogs often execute small actions on behalf of their owner activity, we still attribute the add-listener operation to the activity, and consider whether the activity's lifetime contains the lifetime of listener registration.

▷ *Example:* Consider `DemoLauncher` in the running example. For brevity, let us denote it with a. Since this is the starting activity of the application, we introduce an artificial transition $t_1 = [launcher,a]$ where $launcher$ denotes the Android application launcher. The relevant callbacks are $\sigma(t_1) =$ [`onCreate`,a][`onStart`,a][`onResume`,a]. Next, let $t_2 = [a,a]$ be the transition triggered by button br, with $\sigma(t_2) =$ [`onClick`,br]. Finally, let $t_3 = [a,launcher]$ occur then the hardware BACK button is pressed to close a and go back to the Android launcher. The callbacks are $\sigma(t_3) =$ [`onPause`,a][`onStop`,a][`onDestroy`,a]. Let window transition sequence $T = \langle t_1, t_2, t_3 \rangle$. This sequence is balanced, since the window stack effects are $\delta(t_1) = push\ a$, $\delta(t_2) = [\,]$, and $\delta(t_3) = pop\ a$. Clearly, T represents a lifetime of a. In the application code, the activity defines only `onCreate` (not shown in the figure because it does not have relevant effects) and `onResume`, but not any other lifecycle callbacks. Thus, the relevant callback sequence is [`onCreate`,a][`onResume`,a][`onClick`,br]. Since the last callback contains [`requestLocationUpdates`,l]

```
1  class ListCheckin extends Activity {
2    void onOptionsItemSelected(MenuItem item) {
3      switch(item.getItemId()) { ...
4        case ADD_INCIDENT:
5          Intent i = new Intent(CheckinActivity.class);
6          startActivity(i); return; ... } }
7
8  class CheckinActivity extends Activity
9        implements LocationListener {
10   LocationManager lm = ...;
11   void onCreate() {
12     lm.requestLocationUpdates(this); }
13   void onDestroy() {
14     lm.removeUpdates(this); }
15   void onStart() { ... }
16   void onResume() { ... }
17   void onPause() { ... }
18   void onLocationChanged() {
19     lm.removeUpdates(this); } }
```

Figure 3: Example derived from the Ushahidi application.

and there is no subsequent [removeUpdates,l], window transition sequence T matches Pattern 1. ◁

2.4 Pattern 2: Long-Wait State

Informally, this pattern considers an activity that adds a listener and then is put in a (potentially) long-wait state without removing this listener. We are interested in application states that can suspend the application for arbitrarily long periods of time. Specifically, suppose the the application user presses the hardware HOME button (or, equivalently, selects another application from the list of recent applications). As a result, the current application is put in the background. However, if there are any active location listeners, the battery's energy is still consumed. It may be hours before the user resumes the application. A similar scenario occurs when the hardware POWER button is pressed: the screen is turned off, but active location listeners still drain the battery. Since the callback sequences for these two scenarios are the same, we will discuss only the use of the HOME button to put the application in a long-wait state. This pattern has not been identified in the work on GreenDroid.

Unlike with Pattern 1, here the lifetime of the activity may contain the lifetime of listener registration. This scenario is illustrated by the example in Figure 3. The simplified code in the figure is derived from the Ushahidi application, which was also analyzed in the prior work on GreenDroid. This particular defect is not detected in the experiments from that prior work. Through testing, we have confirmed that indeed this defect drains the battery.

▷ *Example:* Activity `ListCheckin` contains an options menu. Event handler `onOptionsItemSelected` represents the clicking of an item in that menu. One of the menu items is used to open `CheckinActivity`, using the standard Android mechanism of intents. Inside the newly-opened activity, `onCreate` registers the activity as a listener, and `onDestroy` stops the listening. As far as Pattern 1 is concerned, the lifetime of listener registration is contained within the lifetime of the activity that adds the listener. The rest of the lifecycle callbacks defined in the code (lines 15–17) do not have any effect on the listener.

Callback `onLocationChanged` is invoked when a location read is obtained; this method stops the listening. However, it is still possible for the listener to be leaked. If a location read cannot be acquired (e.g., the GPS cannot obtain a satellite fix because of physical obstacles or atmospheric conditions), callback `onLocationChanged` will not be invoked. If at this moment the user presses the HOME or POWER button (e.g., the user gives up after the GPS signal cannot be acquired), the application is put on the background but the listening is still active and is draining the battery. Note that battery drain for such "no-read" listener leaks occurs at the same rate as drain for "normal" listener leaks when location reads are successfully acquired. ◁

To define the general form of this defect pattern, consider a window transition sequence $T = \langle t_1, t_2, \ldots, t_n \rangle$ where $\delta(t_1)$ contains *push* w for an activity w and event $\epsilon(t_n)$ is $[w',home]$ where w' is either the same as w, or is a menu/dialog owned by w. Here w is created at the beginning of T and default event $home$ occurs on w' at the end of T. Suppose also that the sequence of push/pop operations in $\delta(T)$, starting from *push* w on t_1, is valid as defined by non-terminal *Valid* described earlier. Under these conditions, T puts w in a long-wait state.

DEFINITION 3. *Suppose T puts an activity w in a long-wait state. Consider the callback invocation sequence $\sigma(T)$ and its subsequence $s = [c_1,w] \ldots [c_m,w']$ where c_1 is the creation callback for w and c_m is the last callback before the application goes in the background. If s is leaking a listener l, and the corresponding add-listener operation $[a,l]$ occurred when the top-most activity on the window stack was w, then T matches Pattern 2.*

Note that the last callback c_m in this definition is an intermediate point in the invocation sequence for the last transition t_n. If an activity defines all lifecycle callbacks, the entire sequence for t_n is `onPause`, `onStop`, `onRestart`, `onStart`, `onResume`. Callback c_m is `onStop`, since the first two callbacks occur before the application goes in the background, and the last three are executed during re-activation when the user returns to the application.

▷ *Example:* For the example in Figure 3, let us use (for brevity) a to denote `ListCheckin`, m to denote the options menu of a, and a' to denote `CheckinActivity`. Consider the transition sequence $T = \langle t_1, t_2 \rangle$ defined as follows. Transition t_1 is triggered when the `ADD_INCIDENT` menu item is clicked. The window stack effect sequence $\delta(t_1)$ is *pop* m, *push* a'. Here we account for the standard behavior of menus: after a menu item is clicked, the menu is automatically closed—thus, *pop* m should be included in the sequence. Transition t_2 is triggered by event $[a',home]$ and has an empty $\delta(t_2)$. Clearly, T represents a valid sequence of transitions.

In general, sequence $\sigma(t_1)$ is [`onOptionsItemSelected`,ai] [`onOptionsMenuClosed`,m][`onPause`,a][`onCreate`,a'][`onStart`, a'][`onResume`,a'][`onStop`,a]; here ai denotes the menu item with id `ADD_INCIDENT`. Sequence $\sigma(t_2)$ contains [`onPause`,a'][`onStop`, a'][`onRestart`,a'][`onStart`,a'][`onResume`,a']. The last three callbacks in $\sigma(t_2)$ occur after the application is re-activated from the background. Accounting for the subset of callbacks defined in the code, and the definition of Pattern 2, the relevant callbacks are [`onCreate`,a'][`onStart`,a'][`onResume`,a'][`onPause`,a']. The first element in the sequence adds the activity as a listener, but the rest of the callbacks do not remove this listener. Thus, $T = \langle t_1, t_2 \rangle$ matches Pattern 2. A fix for the defect would be to remove the listener in `CheckinActivity.onPause`. ◁

3. Static Detection of Defect Patterns

The run-time behaviors defined earlier can be used as basis for defining static abstractions and static detection analyses based on them. In a minor abuse of notation, for the rest of the paper we will use **Win**, **View**, etc. (Figure 1) to denote sets of static abstractions rather than run-time entities. There are various ways to define such static abstractions. We use the approach from [40, 44], which creates a separate $a \in \mathbf{Win}$ for each activity class, together with appropriate $m, d \in \mathbf{Win}$ for its menus and dialogs, and abstractions $v \in \mathbf{View}$ for their widgets (i.e., defined programmatically or in layout XML files), and then propagates them similarly to inter-procedural points-to analysis, but with special handling of Android API calls.

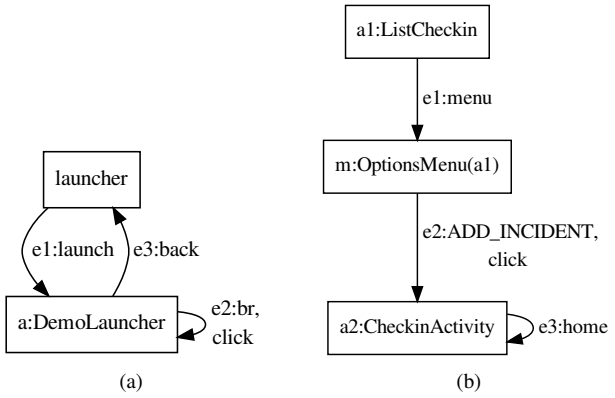

Figure 4: WTGs for the running examples.

Follow-up work [46–48] defines the *window transition graph* (WTG), a static model $G = (\mathbf{Win}, \mathbf{Trans}, \epsilon, \delta, \sigma)$ with nodes $w \in \mathbf{Win}$ and edges $t = [w,w'] \in \mathbf{Trans}$. Each transition t is annotated with trigger event $\epsilon(t)$, callback sequence $\sigma(t)$, and window stack changes $\delta(t)$. Implementations of these analyses are available in our GATOR [41] analysis toolkit for Android, which itself is built using the Soot framework [43] and its Jimple internal representation (constructed either from Java bytecode or from Dalvik bytecode via Dexpler [5]). The energy defect analysis was developed in this infrastructure.

The WTGs for the running examples are shown in Figure 4. These graphs are small because the examples are simplified on purpose, but in actual applications WTGs have hundreds of edges. Given the WTG, our detection analysis proceeds in three phases.

3.1 Phase 1: Add-Listener and Remove-Listener Operations

Consider the set $\{[c,o] \mid t \in G \wedge [c,o] \in \sigma(t)\}$. For each invocation of a callback c on object o, we compute a set $A(c,o)$ of pairs $[a,l]$ of an add-listener API invocation statement a and a listener object l. We also compute a similar set $R(c,o)$ of pairs $[r,l]$. These sets are determined in four steps, as described below.

Step 1 An interprocedural control-flow graph (ICFG) [42] is constructed for c and its transitive callees in the application code. Then, a constant propagation analysis is performed to identify and remove infeasible ICFG edges, based on the knowledge that the calling context of c is o. This analysis is defined in prior work [47], where it was shown to produce more precise control-flow models. For the example in Figure 3, this analysis will determine that when `onOptionsItemSelected` is invoked on the item with id `ADD_INCIDENT`, only one branch of the switch statement is feasible and the rest of the branches can be ignored. In addition, we remove ICFG edges related to null pointer checks and throwing of unchecked exceptions, since in our experience they represent unusual control flow that does not contribute to defect detection.

Step 2 An ICFG traversal is performed starting from the entry node of c. This traversal follows interprocedurally-valid paths. During the traversal, whenever an add-listener API call site a is encountered, the points-to set of the listener parameter is used to construct and remember pairs $[a,l]$. (Points-to sets are derived as described elsewhere [40].) Similarly, we also record all reached pairs $[r,l]$ where r is a remove-listener API invocation statement. For the example in Figure 2, analysis of `onClick` and its callees will identify [`requestLocationUpdates`,`EventManager`] where the second element of the pair denotes the listener created at line 12. For the example in Figure 3, analysis of `onCreate` will identify a similar pair with the activity being the listener. In addition, analysis of `onDestroy` will detect [`removeUpdates`,`CheckinActivity`].

Step 3 For each $[a,l]$ encountered in Step 2, we need to determine whether the ICFG contains a path from statement a to the exit of c along which there does not exist a matching remove-listener operation $[r,l]$. If the ICFG contains at least one $[r,l]$ for the same listener l (as determined by Step 2), we perform an additional traversal on the reverse ICFG, starting from the exit node of c. This traversal considers only valid paths with proper matching of call sites and return sites. During the traversal, whenever a remove-listener API call site r with listener l is encountered and it matches the pair $[a,l]$ being considered, the traversal stops. If the add-listener call site a is never reached, this means that $[a,l]$ is not downward exposed and is not included in set $A(c,o)$. In both of our examples, the add-listener operation is downward exposed and this step does not modify sets $A(c,o)$.

Step 4 We construct a similar set $R(c,o)$ of remove-listener operations. However, only operations that are guaranteed to execute along all possible execution paths should be included in this set. If $[r,l]$ could be avoided along some path, this could lead to a leak of listener l. Thus, for each $[r,l]$ observed in Step 2, we perform a traversal of valid ICFG paths, starting from the entry node of c, and stopping if $[r,l]$ is encountered. If the exit node of c is reached, this means that some valid ICFG path can avoid $[r,l]$. Set $R(c,o)$ excludes such remove-listener operations. In the running example, the analysis of `onDestroy` will determine that each path through the callback must reach the remove-listener call site, and therefore this operation is included in $R(c,o)$.

▷ *Example:* The final sets A and R computed by Phase 1 for the two running examples are as follows:

$A($`onResume`, `DemoLauncher`$) = \emptyset$
$R($`onResume`, `DemoLauncher`$) = \emptyset$
$A($`onClick`, `br`$) =$
{ [`requestLocationUpdates`,`EventManager`] }
$R($`onClick`, `br`$) = \emptyset$
$A($`onOptionsItemSelected`, `ai`$) = \emptyset$
$R($`onOptionsItemSelected`, `ai`$) = \emptyset$
$A($`onCreate`, `CheckinActivity`$) =$
{ [`requestLocationUpdates`,`CheckinActivity`] }
$R($`onCreate`, `CheckinActivity`$) = \emptyset$
$A($`onDestroy`, `CheckinActivity`$) = \emptyset$
$R($`onDestroy`, `CheckinActivity`$) =$
{ [`removeUpdates`,`CheckinActivity`] }
$A($`onStart`/`onResume`/`onPause`, `CheckinActivity`$) = \emptyset$
$R($`onStart`/`onResume`/`onPause`, `CheckinActivity`$) = \emptyset$

Here `br` and `ai` represent the static abstractions of the corresponding run-time widgets. ◁

3.2 Phase 2: Path Generation

The second phase of the analysis creates a set of candidate paths that represents the lifetime of an activity (for Pattern 1) or the transition to a long-wait state (for Pattern 2). For each activity $w \in \mathbf{Win}$, we consider all incoming WTG edges $t_1 = [w',w]$ that have *push* w as the last element in $\delta(t_1)$. Starting from each such t_1, we perform a depth-first traversal to construct "candidate" paths $\langle t_1, t_2, \ldots, t_n \rangle$. The details of this traversal are presented in Algorithm 1. During the traversal, *path* stores the current path and *stack* is the window stack corresponding to that path. We only consider paths whose length does not exceed some analysis parameter k (in our implementation, this parameter's value is 5). Any path that represents a lifetime for the initial activity w or a transition to a long-wait state from w is recorded for later processing.

Algorithm 1: GenerateCandidatePaths

```
1  foreach activity w ∈ Win do
2      foreach edge t1 = [w',w] such that δ(t1) ends with push w do
3          path ← ⟨t1⟩
4          stack ← ⟨w⟩
5          TRAVERSE(w, path, stack)
6  procedure TRAVERSE(w, path, stack)
7      if path.length > k then
8          return
9      if ACTIVITYLIFETIME(path, stack) then
10         record path
11         return
12     if LONGWAIT(path, stack) then
13         record path
14         return
15     foreach edge t = [w,w'] such that t ∉ path do
16         if CANAPPEND(t, path, stack) then
17             DOAPPEND(t, path, stack)
18             TRAVERSE(w', path, stack)
19             UNDOAPPEND(t, path, stack)
```

Helper function ACTIVITYLIFETIME checks the following conditions: (1) $\delta(t_n)$ of the last edge t_n in $path$ contains $pop\ w$, and (2) the stack operations in $\delta(t_n)$, up to and including this $pop\ w$, when applied to $stack$, result in an empty stack. The second condition guarantees that the sequence of push/pop operations from $push\ w$ in $\delta(t_1)$ to $pop\ w$ in $\delta(t_n)$ is a string in the language defined by $Balanced$.

Helper function LONGWAIT determines if $path$ will transit to a long-wait state from the initial activity w. The following conditions are checked: (1) $stack$ is not empty and its top window w' is either w or a dialog/menu owned by w, and (2) the event on the last edge in $path$ is $[w',home]$. Since the window stack is not empty, the sequence of push/pop operations along $path$ is a string in the language defined by $Valid$.

During the depth-first traversal, helper function CANAPPEND (invoked at line 16) considers the sequence $\delta(t)$ of stack operations for a given edge $t = [w,w']$ and decides whether this sequence can be successfully applied to the current window stack. In particular, for each $pop\ w''$ operation in $\delta(t)$, the current top of the stack must match w''. Furthermore, after all operations are applied, the top of the stack must be the same as the target node of t. If CANAPPEND returns true, it means that the sequence of stack push/pop operations in the concatenation of $\delta(path)$ and $\delta(t)$ is a string in the language defined by $Valid$.

If transition t is a valid extension of the current path, helper function DOAPPEND appends t to $path$ and applies stack operations $\delta(t)$ to $stack$. After the traversal of the new path completes, helper function UNDOAPPEND removes t from the path and "unrolls" the changes made to $stack$ due to operations $\delta(t)$.

▷ *Example:* Consider the example in Figure 2 and its WTG shown in Figure 4a. Let a denote the WTG node for `DemoLauncher`. Figure 4a shows transitions $t_1 = [launcher,a]$, $t_2 = [a,a]$, and $t_3 = [a,launcher]$ with events $\epsilon(t_1) = launch$, $\epsilon(t_2) = [br,click]$ and $\epsilon(t_3) = [a,back]$. In addition, consider transition $t_4 = [a,a]$ with $\epsilon(t_4) = [a,home]$ (not shown in the figure). The stack operations for these four edges are $\delta(t_1) = push\ a$, $\delta(t_2) = [\ ]$, $\delta(t_3) = pop\ a$, and $\delta(t_4) = [\ ]$.

For the sake of the example, suppose we execute Algorithm 1 with $k = 3$. The candidate paths for Pattern 1 are $\langle t_1, t_3\rangle$, $\langle t_1, t_2, t_3\rangle$, and $\langle t_1, t_4, t_3\rangle$. The second path corresponds to the problematic leaking behavior, as discussed earlier. The candidate paths for Pattern 2 are $\langle t_1, t_4\rangle$ and $\langle t_1, t_2, t_4\rangle$. For the second path, the callback sequence before the application goes in the background is $[$`onResume`$,a][$`onClick`$,br]$ (because no other lifecycle callbacks are defined in the application), and therefore this is also a leaking path. The next phase of the analysis considers all these candidate paths and identifies the ones with leaking callback sequences. ◁

3.3 Phase 3: Detection of Leaking Callback Sequences

In this phase, we perform leak detection on candidate paths recorded in Phase 2. First, the relevant callback sequence is extracted from each candidate path. Consider a transition sequence $T = \langle t_1, \ldots, t_n\rangle$ which represents a Pattern 1 candidate path. The relevant callback subsequence of $\delta(T)$ is $[c_1,w] \ldots [c_m,w]$ where c_1 is the creation callback for w and c_m is the termination callback for w (w is the target window of edge t_1). Similarly, for a sequence $T = \langle t_1, \ldots, t_n\rangle$ which is a Pattern 2 candidate path, the relevant subsequence is $[c_1,w] \ldots [c_m,w']$ where c_1 is the creation callback for w and c_m is the last callback before the application goes in the background.

Given a sequence of callbacks $s = [c_1,o_1][c_2,o_2] \ldots [c_m,o_m]$, we consider its sequence $A_1, A_2, \ldots, A_m$ of add-listener sets and $R_1, R_2, \ldots, R_m$ of remove-listener sets. Recall from Definition 1 that s leaks listener l if there exists an add-listener operation $[a,l] \in A_i$ such that for each $j > i$ there does not exist a matching $[r,l] \in R_j$. In Phase 1, we have already computed sets $A(c, o)$ and $R(c, o)$ for any relevant c and o. To detect leaks, we examine each element $[c_i,o_i]$ of s in order and maintain a set L of added but not yet released listeners. Initially, L is empty. When $[c_i,o_i]$ is processed, all elements of $R(c_i, o_i)$ are removed from L, and then all elements of $A(c_i, o_i)$ are added to L. Any $[a,l]$ that remains in L at the end of this process is considered to be a leak.

▷ *Example:* Consider again Figure 2 and the WTG in Figure 4a. We have $t_1 = [launcher,a]$, $t_2 = [a,a]$ for the button click, $t_3 = [a,launcher]$ for $back$, and $t_4 = [a,a]$ for $home$. Candidate paths for Pattern 1 (for $k = 3$) are $\langle t_1, t_3\rangle$, $\langle t_1, t_2, t_3\rangle$, and $\langle t_1, t_4, t_3\rangle$. For the first and the third path, the relevant callback sequence is $[$`onResume`$,a]$ which is not leaking because both $A($`onResume`$, a)$ and $R($`onResume`$, a)$ are empty. For the second path, callback sequence $[$`onResume`$,a][$`onClick`$,br]$ leaks $[$`requestLocationUpdates`$,$`EventManager`$]$. For candidate path $\langle t_1, t_2, t_4\rangle$ for Pattern 2, the callbacks before the application goes in the background are also $[$`onResume`$,a][$`onClick`$,br]$ and there is a leak as well. ◁

Defect reporting For any leaking candidate path $\langle t_1, \ldots\rangle$, the analysis records the pair $[w,l]$ of the initial activity w (i.e., the target of t_1) and the leaking listener l, identified by the allocation site of the corresponding object. For Figure 2, this would be $[$`DemoLauncher`$,l_{12}]$ where l_{12} is the `EventManager` allocation site at line 12 in the code. For each recorded pair, the leaking candidate paths for that pair are also recorded. Each $[w,l]$ is reported as a separate defect, since it requires the programmer to examine the callbacks associated with w and to determine whether they manage listener l correctly.

Defect prioritization In addition to these reports, we also classify leaking listeners as "high" or low "low" priority, based on the following rationale. Consider again the example in Figure 3. The leaking behavior can be observed only when a location read is not obtained (e.g., the weather does not allow a GPS fix), which arguably is not a very frequently-occurring situation. If we analyze `onLocationChanged`—the callback that is executed on a listener l when a location read is obtained—we can determine whether it contains a remove-listener operation for l along each execution

Application	WTG			Defects				Time
	Nodes	Edges	Paths	Pat-1	Real-1	Pat-2	Real-2	(s)
droidar	10	120	82292	2	2	2	2	2.47
osmdroid	14	92	1425	0	0	2	2	0.07
recycle	7	22	98	1	1	1	1	0.04
sofia	11	55	237	1	1	1	1	0.05
ushahidi	42	296	31416	1	1	3	3	1.21
droidar-f	10	120	82292	1	1	1	1	2.44
osmdroid-f	14	92	1425	0	0	0	0	0.07
recycle-f	8	29	258	0	0	0	0	0.04
sofia-f	15	67	406	0	0	0	0	0.08
ushahidi-f	42	284	30758	0	0	2	2	0.71
heregps	3	14	414	1	1	1	1	0.05
locdemo	5	13	228	1	1	1	1	0.04
speedometer	2	5	10	1	1	1	1	0.03
whereami	5	17	51	1	1	1	1	0.03
wigle	18	64	3769	1	0	1	1	0.41

Table 1: Analyzed applications and detected defects.

path. If this is the case, a location read will release the listener. In the defect report from the analysis such listeners are labeled as "low priority": they should still be examined by the programmer, but perhaps after other leaking listeners have been examined. To make this distinction, for each leaking l we analyze the corresponding callback (`onLocationChanged` or similar method) using the same approach as in Step 4 of Phase 1. The defect in Figure 3 will be reported as low priority, while the one from Figure 2 will be high priority. In our experiments 3 out of the 17 reported defects were classified as low priority ones.

4. Evaluation

The static analysis was implemented in GATOR [41], our open-source static analysis toolkit for Android. The toolkit contains implementations of GUI structure analysis [40, 44] and WTG generation [46–48]. The implementation of the energy defect analysis is currently available as part of the latest release of GATOR.

The goal of our evaluation is to answer several questions. First, how well does the analysis discover GUI-related energy-drain defects already known from prior work? Second, does the analysis discover defects that have not been identified in prior work? Third, does the detection exhibit a reasonably small number of false positives? Finally, what is the cost of the analysis?

Benchmarks To answer these questions, we used several sources of benchmarks, as listed in Table 1. First, we considered the benchmarks from the work on GreenDroid [27] that exhibit defects due to incorrect control flow and listener operations in the UI thread of the application. Almost all such GUI defects involve operations related to location awareness, and our static analysis was built to track add/release operations for location listeners. We also considered the fixed versions of these benchmarks—that is, the versions that involve fixes of these known defects. Both defective and fixed versions were obtained from the public GreenDroid web site.[3] In Table 1, applications in the first part of the table are the defective ones, while applications in the second part of the table, suffixed with `-f`, are the ones with defect fixes.

We also considered the F-Droid repository of open-source applications[4] and searched for applications that use location-awareness capabilities in their UI-processing code. Specifically, the textual description and the manifest file were checked for references to location awareness of GPS, and the code was examined to ensure that the UI thread uses location-related APIs. For the applications we could successfully build and run on an actual device, the static analysis was applied to detect potential defects. Out of the 10 applications that were analyzed, 5 were reported by the analysis to contain defects. The last part of Table 1 shows these 5 applications.

Columns "Nodes" and "Edges" show the numbers of WTG nodes and edges, respectively. Column "Paths" contains the number of candidate paths that were recorded and then analyzed for leaking listeners. The last column in the table shows the cost of the analysis; for this collection of experiments, this cost is very low.

Detected defects Recall that for a leaking path $\langle t_1, \ldots \rangle$, the analysis reports a pair $[w,l]$ of the initial activity w (i.e., the target of t_1) and the leaking listener l. We consider each $[w,l]$ to be a defect. Column "Pat-1" shows the number of such defects that were reported by the static analysis as instances of Pattern 1. Column "Pat-2" shows a similar measurement for Pattern 2. In our experiments, a total of 17 unique pairs $[w,l]$ were reported, and all defects that match Pattern 1 (11 defects) also match Pattern 2 (17 defects), but not vice versa. However, it is still useful to detect both patterns statically, as they correspond to two different scenarios. If a defect matches both Pattern 1 and Pattern 2 (e.g., the one in Figure 2), it usually means that the programmer completely ignored the removal of the listener. On the other hand, if a defect matches Pattern 2 but not Pattern 1 (e.g., the one in Figure 3), this means that the programmer attempted to remove the listener, but did not do it correctly. Given the low cost of the analysis, we believe that detection of both patterns is valuable.

Column "Real-1" shows the number of detected defects from column "Pat-1" that we manually confirmed to be real, by observing the actual run-time behavior of the application. Similarly, column "Real-2" shows the number of defects from column "Pat-2" that were verified in the same manner. Only one reported defect is incorrect: in `wigle`, a defect is incorrectly reported by the analysis as an instance of both Pattern 1 and Pattern 2, while in reality it is only an instance of Pattern 2. The cause of this imprecision will be discussed shortly.

Two conclusions can be drawn from these measurements. First, the analysis successfully detects various defects across the ana-

[3] `sccpu2.cse.ust.hk/greendroid`

[4] `f-droid.org`

Application	$\|D\|$	$\|S\|$	$\|D-S\|$	$\|S-D\|$
`droidar`	1	2	0	1
`osmdroid`	2	2	0	0
`recycle`	1	1	0	0
`sofia`	1	1	0	0
`ushahidi`	1	3	0	2

Table 2: Defects reported: GreenDroid (D) vs static analysis (S).

lyzed applications. Even the "fixed" versions are not free of defects: for example, we discovered two defects in `ushahidi-f` that were not reported in the work on GreenDroid, and were missed by the application developers when `ushahidi` was fixed to obtain `ushahidi-f` (in fact, these two defects are quite similar to the one that was fixed). A similar situation was observed for `droidar`. This observation indicates the benefits of static detection, compared to run-time detection which depends on hard-to-automate triggering of the problematic behavior. Of course, static detection has it own limitations, with the main one being false positives. However, the experimental results for the 15 benchmarks shown in Table 1 indicate that the proposed analysis achieves very high precision.

False positive The false positive for `wigle` is because the developer decided to override standard method `Activity.finish` with a custom version which removes the listener. When method `finish` is invoked on an activity (by the application code or by the framework code), this causes the termination of the activity. However, this method is not a callback that is defined as part of the lifecycle of an activity, and is rarely overridden by applications. In other words, `finish` can be called to force termination, but it is not executed as part of the actual termination process. Thus, `finish` does not appear on WTG edges (although it is accounted for during WTG creation [48]). In fact, termination could happen even if `finish` is not called: for example, the system may silently terminate an activity to recover memory [14]. The Android lifecycle model guarantees that `onDestroy` will be called in all scenarios, and this is where the listener should be removed, rather than in `finish`. This example indicates that the developer misunderstands the activity lifecycle. During the manual examination of this defect on a real device we did observe that the location listener is properly released, and decided to classify the defect as a false positive, although one could argue that it violates Android guidelines.

Comparison with GreenDroid To compare the proposed static detection with the most relevant prior work, Table 4 considers the UI thread defects $[w,l]$ reported by the dynamic analysis approach in GreenDroid. For a given application, let D be the set of these defects, while S be the set of defects reported by our static analysis. The sizes of these sets are given in the second and third column in Table 4. The next two columns show the sizes of sets $D-S$ and $S-D$, respectively. As the next-to-last column shows, our static analysis reported *all* defects from the prior dynamic analysis work. The last column shows how many of the statically-detected defects were *not* reported by GreenDroid; for two of the applications, there are additional defects we discovered statically (and these defects are still present in the fixed versions from GreenDroid). A possible explanation of this result is that the run-time exploration strategy in GreenDroid did not trigger the necessary GUI events; in general, comprehensive run-time GUI coverage is challenging [8].

Overall, these results indicate that static detection could be more effective than dynamic detection. At the same time, it is important to consider the relative strengths and weaknesses of both approaches: while the static analysis can model more comprehensively certain behaviors of the UI thread, other aspects of run-time semantics are not modeled statically (e.g., asynchronous tasks and services) and dynamic analysis does capture additional defects for such behaviors. This highlights the need for more comprehensive static control-flow analyses for Android, as well as hybrid static/dynamic approaches for defect detection.

Summary On 15 Android applications, the proposed analysis exhibits low cost and high precision. It discovers all GUI-related leaking-listener defects discovered by GreenDroid, as well as three new ones. Additional defects were discovered in 5 applications not analyzed in prior work. With one exception, all reported defects are observable at run time. These results indicate that the analysis is suitable for use in real-world static checking tools.

5. Related Work

Energy analysis for Android There is a growing body of work on analyzing and optimizing the energy consumption of mobile devices. Studies have shown [6, 25] that battery drain issues on an Android device could be caused by poorly managed background activities and inappropriate invocations to energy-related APIs. Qian et al. [38] developed a resource usage profiler to uncover inefficient usage of radio and network resources. Some energy optimizers [22, 26] could reduce the energy consumption when an application is displayed on OLED screens, by modifying the color scheme; these techniques do not consider any energy-related defects. Oliner et al. [33] used statistical battery discharge rate data of smartphone applications in a community of devices in order to uncover battery draining applications. A similar approach is used by Min et al. [30]. Though this method reveals accurate results, it requires runtime data collected from a large number of devices that have the target application installed, which is not always possible.

An approach from Zhang et al. [49] uses dynamic taint analysis to detect design flaws related to network operations. However, this approach introduces relatively high run-time overhead and cannot be used to analyze other energy defects. Early work of energy-aware profiling [19, 21, 34, 35] used dynamic analysis to detect energy hotspots in Android applications. However, as with other program profiling techniques, they require comprehensive test cases to execute the application on the device. Furthermore, an energy hotspot may not always point to the underlying energy-related defect. For example, if there is an energy leak due to mismanagement of lifecycle callbacks, these callbacks (which are typically short lived) are unlikely to be reported as hotspots.

Following their work on energy profiling [34, 35], Pathak et al. defined a static analysis to detect energy-related defects [36]. This analysis aims to model control flow along multiple threads of execution and to detect code paths for which energy-related resources (e.g., location listeners and wake locks) are not properly released. The control-flow modeling is very simplistic: while some simple sequences of lifecycle callbacks are considered in the analysis, there is no modeling of interleaving of callbacks across multiple windows, nor is there modeling of the window stack. Furthermore, the approach requires human involvement: the developer is expected to specify expected invocation orders of widget event handlers, which is impractical and may lead to limited coverage of possible behaviors. In our approach, the order of callbacks in the UI thread is determined automatically through a more general and precise analysis. The vast majority of defects in this work (e.g., wake lock leaks) are reported for service components running concurrently with the UI thread. There are only two applications for which location-awareness defects are reported. One defect is in `ushahidi` (already discussed earlier); the same defect is discovered by GreenDroid and by our approach, but we also detect two other similar defects not reported by [36] or by GreenDroid. The other case is an application whose code is not available anymore (dead URL) and

whose version number is not specified. In future work, it would be interesting to restate the entire approach by Pathak et al. as a fully-automated analysis of multiple threads of execution, accounting for the advances in control-flow analysis for Android that have been developed since 2012 when this early work was published.

Banerjee et al. [4] developed an energy bug and hotspot detection tool based on dynamic analysis. In their work, they use a modified version of Dynodroid [29], an Android event input generation system, to generate a model of the GUI. The model is then used to trigger sequences of GUI events which are profiled for energy usage. Event sequences that match certain heuristic conditions are reported as defects or hotspots. As recent work shows [8], achieving high run-time coverage for Android applications is a difficult problem because of the event-driven and framework-based control flow and data flow. Furthermore, defect detection is based on heuristics utilizing run-time measurements, and does not identify the callback sequences or code locations that are responsible for the actual defect. Static analysis such as ours can be more comprehensive and can be more easily deployed in the development environment because it does not require the physical measurements of voltage and current needed for energy profiling in [4]. On the other hand, as with any dynamic analysis, one advantage is the potentially higher precision due to the available run-time information; in contrast, static analysis may exhibit false positives. This prior work reports one location listener leaked by an activity, but our static analysis does not report this defect. We examined the code and confirmed that this is indeed not a defect: there is no execution that would trigger a leak, and it is unclear why [4] reports it.

GreenDroid [27] uses Java PathFinder (JPF) to trigger various GUI event sequences based on a model of GUI structure and behavior. The behavioral model is manually extracted from Android specifications, but it does not provide the same level of precision and detail as our WTG representation. During run-time exploration, lifetime constraints similar to our Pattern 1 are checked, together with dynamic information flow measurements to track whether sensory data is underutilized. The applications that exhibited UI leaks of location listeners in their studies were also used in our experiments, as shown in Tables 1 and 4. All such defects were successfully detected statically. Furthermore, even after their bug fixes, we discovered three additional defects. On the other hand their approach detects leaks in services, which are not modeled by our static analysis. Similarly to the dynamic analysis from [4], the challenges of obtaining comprehensive run-time coverage of relevant behaviors presents an obstacle for this technique, with further complications due to JPF limitations. As usual, static analysis (exemplified by our low-cost, high-precision technique) allows more comprehensive reasoning about possible run-time behaviors. An interesting direction for future work is to develop a hybrid approach which uses static analysis to detect conservatively a set of potential defects and to focus subsequent dynamic exploration on the corresponding run-time behaviors.

Static control-flow analysis for Android A significant body of work on security analysis [7, 9, 11, 17, 20, 23, 28, 32, 51] uses static modeling of certain aspects of Android control flow (and the related data flow). These approaches do not provide a comprehensive model of GUI control flow and the possible sequences of callbacks. A related static analysis [3] constructs an activity transition graph (used for run-time GUI exploration) but without capturing the general GUI effects of callbacks and the window stack changes. Other static analyses also model the sequences of callbacks in Android, for the purposes of race detection (e.g., [24]), leak analysis (e.g., [18]), and static checking (e.g., [37, 50]); all these approaches employ ad hoc control-flow modeling that lacks generality.

FlowDroid [2] uses an artificial main method to represent the effects of callbacks from the Android framework to the application code. This main method encodes possible sequences of callbacks, but does not account for the full generality of GUI effects of event handlers, and does not represent precisely the interleaving of callbacks that span multiple activities and their lifetimes. We developed a more comprehensive solution [46–48] which encodes the relevant control-flow information in the window transition graph described earlier. This work also defined the modeling of the window stack. In addition to serving as basis for the defect analysis presented here, the WTG representation and traversals could potentially be useful for other static detection techniques that require control-flow analysis of event/callback-driven Android behavior, perhaps by adapting more general typestate analyses [10, 31].

6. Conclusions

We propose a static analysis for detection of energy-related defects in the UI logic of an Android application. The technical foundation for this analysis is the static modeling of possible sequences of window transitions and their related callbacks. By identifying certain such sequences, based on the state of the window stack, we define precisely two patterns of behavior in which location listeners are leaking. Control-flow analysis of individual callbacks is combined with analysis of callback sequences to identify instances of these patterns. Seventeen known and new defects were detected in previously-analyzed and never-analyzed applications. All but one of the reported defects are observable at run time. The evaluation also shows that the cost of the analysis is low.

Two directions for future work are natural extensions of our current approach. First, the general pattern for control-flow analysis is not specific to energy-related defects. A variety of other defect patterns can be defined, based on some notion of "interesting" WTG paths and relevant operations inside the callbacks along such paths. For example, one could consider resource leaks (e.g., for heap memory, native memory, JNI references, or threads [45]) or more general typestate-like properties [10, 31] that need to be tracked across sequences of callbacks. Second, the control-flow analysis could be generalized to represent multiple threads running concurrently with the main UI thread, both for general Java threads and for Android-specific constructs such as asynchronous tasks and services. Such generalizations can be used as basis for static detection of both energy-related defects and other categories of problematic patterns of behavior.

Acknowledgments

We thank the CC 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] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, pages 259–269, 2014.

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

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

[5] A. Bartel, J. Klein, Y. L. Traon, and M. Monperrus. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *SOAP*, 2012.

[6] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *MobiCom*, pages 40–52, 2015.

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

[8] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? In *ASE*, pages 429–440, 2015.

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

[10] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, pages 133–144, 2006.

[11] 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.

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

[13] Google. Tasks and back stack. `developer.android.com/guide/components/tasks-and-back-stack.html`, 2015.

[14] Google. Managing the activity lifecycle. `developer.android.com/training/basics/activity-lifecycle`.

[15] Google. Android dialogs. `developer.android.com/guide/topics/ui/dialogs.html`, 2015.

[16] Google. Location strategies. `developer.android.com/guide/topics/location/strategies.html`, 2015.

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

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

[19] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE*, pages 92–101, 2013.

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

[21] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for Android applications. In *ISSTA*, pages 78–89, 2013.

[22] D. Li, A. H. Tran, and W. G. J. Halfond. Nyx: A display energy optimizer for mobile web apps. In *FSE*, pages 958–961, 2015.

[23] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, P. Aldous, and D. Van Horn. Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation. In *SPSM*, pages 21–32, 2013.

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

[25] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: An empirical study. In *MSR*, pages 2–11, 2014.

[26] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Optimizing energy consumption of GUIs in Android apps: A multi-objective approach. In *FSE*, pages 143–154, 2015.

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

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

[29] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *FSE*, pages 224–234, 2013.

[30] C. Min, Y. Lee, C. Yoo, S. Kang, S. Choi, P. Park, I. Hwang, Y. Ju, S. Choi, and J. Song. PowerForecaster: Predicting smartphone power impact of continuous sensing applications at pre-installation time. In *SenSys*, pages 31–44, 2015.

[31] N. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA*, pages 347–366, 2008.

[32] 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.

[33] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *SenSys*, pages 10:1–10:14, 2013.

[34] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *EuroSys*, pages 153–168, 2011.

[35] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? In *EuroSys*, pages 29–42, 2012.

[36] 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*, pages 267–280, 2012.

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

[38] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: A cross-layer approach. In *MobiSys*, pages 321–334, 2011.

[39] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

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

[41] 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/gator`.

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

[43] 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*, pages 18–34, 2000.

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

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

[46] S. Yang. *Static Analyses of GUI Behavior in Android Applications*. PhD thesis, Ohio State University, Sept. 2015.

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

[48] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for Android. In *ASE*, pages 658–668, 2015.

[49] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. Dinda, and L. Yang. ADEL: An automatic detector of energy leaks for smartphone applications. In *CODES+ISSS*, pages 363–372, 2012.

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

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