Automated Test Generation for Detection of Leaks in Android Applications

Hailong Zhang
Ohio State University

Haowei Wu
Ohio State University

Atanas Rountev
Ohio State University

ABSTRACT

Android devices have limited hardware resources (e.g., memory). Excessive consumption of such resources may lead to crashes, poor responsiveness, battery drain, and negative user experience. We propose an approach for systematic automated test generation to expose resource leak defects in Android applications. We first define the notion of a neutral sequence of GUI events. Intuitively, such a sequence can be expected to have "neutral" effects on resource consumption, and repeated executions of this sequence should not exhibit resource growth. Using a state-of-the-art static control-flow model of an Android application, we demonstrate how to achieve automated generation of such sequences. We then define test generation algorithms for two important categories of neutral sequences, based on common leak patterns specific to Android. Our experimental evaluation compares this approach with a non-automated approach from prior work. The results from this evaluation strongly indicate that it is possible to achieve effective, general, and automated test generation for resource leaks in Android applications using the proposed techniques.

1. INTRODUCTION

The market for Android devices has experienced exponential growth during the past few years [8]. Aiming to provide long-lasting yet portable daily computing services, smartphones are usually equipped with low-power electronic components and limited hardware resources. For example, in Android's Dalvik virtual machine (VM) the available heap memory for an application typically ranges from 16 MB to 192 MB depending on device specifications. In contrast, in traditional laptops and PCs there are hundreds even thousands of MB available in the heap of VMs. Other examples of limited resources include bitmaps, binders (used by Android's inter-process communication mechanism) and threads. Excessive consumption of such resources may lead to crashes, poor responsiveness, battery drain [26], and negative user experience.

AST'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2896921.2896932

The target of our work is *systematic automated test generation to expose resource leak defects in Android applications*. For such applications, the management of resources is challenging. Google's training and best-practices documents [10, 11] offer numerous suggestions to help developers avoid common pitfalls in resource management and usage, e.g., resource leaks where applications fail to release resources appropriately. For software engineering researchers, these guidelines provide an opportunity to automate the detection and prevention of resource leak defects.

Automated testing for Android has seen significant advances over the last few years; a recent study by Choudhary et al. [6] summarizes some of these efforts. However, testing for resource leaks has not been investigated extensively. Random testing approaches (e.g., [12, 16, 19]) cannot be expected to trigger the repeated behaviors needed to observe leaks. Targeted exploration/testing approaches (e.g., [1, 2, 3, 4, 15, 17, 20, 25, 32] are primarily interested in achieving high coverage of possible application behaviors. Our prior work [28] considers testing for resource leaks, but relies on manual control-flow modeling and employs simplistic test selection criteria that lack generality, require application knowledge, and are not automated.

Our proposal. We propose an automated approach for test generation and execution, based on the following novel contributions. First, we define the notion of a *neutral sequence of GUI events*. Suppose the current activity is a (activities are the main components of Android applications). A neutral sequence triggers a series of window-open and window-close operations that (1) balance each other out—that is, each newly-opened window is closed, (2) a may be replaced with another instance a' of the same activity class, and (3) other windows that exist before the sequence starts are not affected. Intuitively, such a sequence can be expected to have "neutral" effects on resource consumption, and repeated executions of this sequence should not exhibit resource growth. While an earlier notion of neutral sequences was considered in our prior work [28], in this paper we provide a much more general and precise definition which, unlike the one from [28], can be used for *systematic* and *automated* generation of neutral sequences and the corresponding test cases, with the help of a general static control-flow analysis for Android.

The second contribution of our approach is a test generation algorithm that targets two important categories of neutral sequences. The first category considers repeated executions of neutral sequences for one activity a and its related "helper" windows: options menus, context menus, and di-

alogs. This category is based on the observation that leaks often are caused by the mismanagement of resources during the lifecycle of an activity [13]. The second category includes sequences that capture the interaction of two activities a and a' (and their related menus and dialogs); this is another common leak pattern. Our static analysis algorithms systematically explore the space of possible neutral sequences for these two categories, using a general state-of-the-art static model of an application's GUI.

The last contribution of our work is a series of experiments that compare the leak detection of the proposed automated approach with the manual approach from our prior work [28]. In that earlier work, application-specific knowledge was needed for several aspects of the test generation. We demonstrate that the new automated test generation has comparable performance—it discovers 16 out of the 18 leak defects detected in [28], as well as 2 new defects that were not detected in the prior work. These results strongly indicate that it is possible to achieve effective, general, and automated test generation for resource leaks in Android applications using the proposed techniques.

2. BACKGROUND

This section describes the GUI semantics that drives the application's behavior and then defines the notion of "neutral" event sequences used by the proposed test generation.

2.1 Relevant Android GUI Features

Windows and views. *Activities* are instances of subclasses of `android.app.Activity`. They are the major application components and correspond to GUI windows that interact with the user through a set of GUI *widgets*. The widgets are referred to as "views" in Android terminology.

We also consider the windows for dialogs and menus. *Dialogs* are objects from subclasses of `android.app.Dialog`. *Menus* are associated with activities ("options" menus) and widgets ("context" menus); they instantiate classes that implement `android.view.Menu`. A dialog or a menu is intended for a short interaction with the user on behalf of the activity that is currently displayed (the *owner activity* for that dialog or menu). Dialog/menu lifetime is contained within the lifetime of its owner activity. For each activity a, let the *cluster* of a, denoted by $cluster(a)$, be the set containing a and all menus and dialogs owned by a. Let **Win** denote the set of all run-time windows and **View** denote the set of all run-time widgets in these windows.

▷ *Example:* Figure 1 shows an example derived from a real-world leak in `connectbot` described in [28]. Class `HostListActivity` defines an activity. Methods `onCreate` and `onDestroy` are lifecycle callbacks (discussed shortly) called by the framework when the activity starts and terminates. Field `list` at line 4 is a list widget for displaying a sequence of list item widgets (for remote SSH hosts). The call at line 5 attaches a listener for "click" events on list items. The call at line 7 allows a context menu to be associated with a list item. When the user performs a "long click" event on an item, an instance of `ContextMenu` is created, `onCreateContextMenu` is invoked on it, and a menu item `edit` for editing the selected SSH host is added to the context menu. Such menu items will be referred to as "views" (although they are not instances of `View`). Field `sort` is also a menu item used for sorting the elements in `list`. If the user presses the hardware MENU

```
 1  public class HostListActivity extends Activity {
 2    private MenuItem sort;
 3    public void onCreate() { ...
 4      ListView list = this.getListView(); ...
 5      list.setOnItemClickListener(
 6        new OnItemClickListener() { ... });
 7      registerForContextMenu(list); ... }
 8    public void onDestroy() { ... }
 9    public boolean onCreateOptionsMenu(Menu menu) {
10      sort = menu.add(R.string.list_menu_sortname);
11      sort.setOnMenuItemClickListener(
12        new OnMenuItemClickListener() { ... }); ... }
13    public void onCreateContextMenu(ContextMenu menu) {
14      MenuItem edit = menu.add(R.string.list_host_edit);
15      edit.setOnMenuItemClickListener(
16        new OnMenuItemClickListener() {
17          public boolean onMenuItemClick(MenuItem item) {
18            Intent i = new Intent(HostEditorActivity.class);
19            startActivityForResult(i, REQUEST_EDIT);
20            return true; } }); ... } ... }

21  public class HostEditorActivity extends Activity { ... }
```

Figure 1: Example derived from `connectbot` application.

button, the framework will create an options menu for the activity and add `sort` to it (line 10). The owner activity for the two menus is `HostListActivity`, and thus they form a cluster together with it.

One of the test cases we generate will long-click on a list item, then click the `edit` item in the context menu, and finally click the hardware BACK button to get back to the list display. When this sequence of three events is repeated several times, the application crashes. Every time `HostListActivity` is (re)entered, a background service is started and a new binder object is returned. When the user leaves the activity, the service is unbound. However, the binder object is kept alive by references in the framework and cannot be garbage collected. This leak is related to a documented problem in the framework code.[1] ◁

Events. A window can respond to *widget events* on the views contained in this window. For such events we will use the notation $e = [v,k]$ where $v \in$ **View** is a widget and k is an event kind. For the example in Figure 1, we have events $[li,click]$ and $[li,long\_click]$ where li is the list item widget on which the event was triggered by the user. We also have events $[edit,click]$ and $[sort,click]$ to represent the clicking on the menu item *edit* (from the context menu) and *sort* (from the options menu).

In addition to widget events, we need to consider *default events*. Pressing the hardware BACK button is represented by event *back*. Event *rotate* indicates that the user rotates the screen, which recreates the current activity with a different layout. Event *home* abstracts an interaction in which the user presses the HOME button and then later resumes the application. Event *power* corresponds to a scenario in which the POWER button is used to turn off the screen, and then the device is reactivated. As discussed earlier, event *menu* indicates that the hardware MENU button was used to open an options menu. We will use the notation $e = [w,k]$ for default events; here w is the window that is currently active and interacting with the user. In Figure 1, let a denote a window of `HostListActivity`. There are five default events $[a,\ldots]$. Event $[a,menu]$ creates the options menu m associated with a. For this menu we have events $[m,\ldots]$ for *back*, *home*, *power* and *rotate*. Let **Event** denote the set of all

[1] `code.google.com/p/android/issues/detail?id=6426`

widget events and default events in the application.

Callbacks. When an event $e \in$ **Event** occurs, it causes a sequence of *callback invocations* $[c_1,o_1][c_2,o_2]\ldots[c_m,o_m]$. Here c_i is a callback method defined in the application code, and o_i is a run-time object (either a widget or a window). Each of these invocations completes before the next one starts.

Widget event handler callbacks respond to widget events; an example is `onMenuItemClick` in Figure 1. *Lifecycle callbacks* are used to manage the lifetime of activities, menus, and dialogs. For example, in Figure 1, callback `onCreate` and `onDestroy` indicate the creation and destruction of the activity. The handling of default events also results in callback invocations: e.g., [a,*home*] for an activity a triggers `onPause`, `onStop`, `onRestart`, `onStart`, `onResume` on a.

▷ *Example:* In Figure 1, consider widget event [li,*click*] on some list item widget li. This event will trigger a callback invocation [`onItemClick`,li]. Similarly, consider [li,*long_click*]. The Android framework defines an internal event handler for this event; this handler creates a `ContextMenu` instance m and triggers [`onCreateContextMenu`,m]. Event [*edit*,*click*] triggers [`onMenuItemClick`,*edit*] [`onPause`,a_1] [`onCreate`,a_2] [`onStart`,a_2] [`onResume`,a_2] [`onStop`,a_1] where a_1 and a_2 are objects of `HostListActivity` and `HostEditorActivity`. ◁

Window transitions. A *window transition*, denoted by a pair $t = [w,w'] \in$ **Win** $\times$ **Win**, indicates the scenario that a new window w' (possibly the same as w) becomes active by performing a GUI event on the previously active window w. Accordingly, for each transition t, we use $\epsilon(t) \in$ **Event** and $\sigma(t)$ to denote the event that causes t and the sequence of callback invocations that occur as part of t. We model the behavior of opening and closing windows of transition t by a sequence of window push/pop operations performed on a *window stack* which stores windows that are currently active. We denote these effects by $\delta(t) \in (\{push, pop\} \times$ **Win**$)^*$. Some behaviors cannot be modeled with a stack, e.g., the launch modes for activities, but they are rarely used [27] and thus not considered in this work.

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 invocation sequences $\sigma(t_i)$. Similarly, let $\delta(T)$ be the concatenation of window stack update sequences $\delta(t_i)$. In general, these effects could involve several windows and can trigger complicated callback sequences.

2.2 Neutral Sequences of Window Transitions

Given a window stack in which the top element is an activity a, a *neutral sequence of transitions* T is one that leaves the stack in the same state, except for possibly replacing a with another instance a' of the same activity class. If a' is the same instance as a, the sequence opens and closes new windows, but never affects a or the windows that are below a in the window stack. If a' is a different instance than a, then the execution of the sequence destroys a and replaces it with a', opens/closes other windows on top of a and a', but does not affect the stack elements below a.

Formally, let $T = \langle t_1, t_2, \ldots, t_n\rangle$ be such that the target of t_i is the same as the source of t_{i+1}, the source of t_1 is a, and the target of t_n is a' where a' is an instance of the same activity class as a (it is possible that a' is the same as a). T is a neutral transition sequence if the stack updates $\delta(T)$ can be successfully applied to a stack containing only a, resulting in final stack containing only a'. Equivalently,

Definition 1. *T is a neutral transition sequence if the string $(push\ a) \circ \delta(T)$ is an element of the language*

$$L \rightarrow Balanced\ push\ a'\ Balanced$$

where

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

Here $\circ$ denotes concatenation. *Balanced* is the standard language [22] that describes sequences of matching push and pop operations—usually used to model balanced updates to the *call stack*, but in our case modeling the *window stack*.

Intuitively, a neutral sequence represents a period of time during which several new windows are created and destroyed, without affecting windows that already exist at the start of the period (except possibly for a). Such a sequence represents a potential target for test generation: if a test case contains repeated executions of such a sequence, it may be able to expose leaking behaviors. Since leaks sometimes involve the destruction and re-creation of the current activity (i.e., the activity a that is on top of the window stack when the sequence starts), our definition allows a to be replaced by a'. While prior work has defined a similar idea of a "neutral cycle" [28], that work (1) considers only a simplistic manually-created model of possible window transitions, (2) does not consider the push/pop effects on the window stack due to the transitions, and thus incorrectly models a variety of run-time behaviors, and (3) considers only limited categories of neutral sequences.

▷ *Example:* Sequence $T = \langle t_1, t_2, t_3\rangle$ is a neutral transition sequence that corresponds to a real leak in this application (also see Figure 2 which illustrates a static abstraction of the related window transitions). Transition t_1 is triggered by a long-click event on a list item in `HostListActivity`. This event creates a context menu attached to the selected item. Next, t_2 is the transition from the context menu to an instance of `HostEditorActivity` by clicking *edit* context menu item, and t_3 is the transition due to the default *back* event from `HostEditorActivity` back to `HostListActivity`. Transitions t_1, t_2, and t_3 are illustrated by edges e_8, e_{14}, and e_{15} in Figure 2. Given a stack in which the top element is `HostListActivity`, this sequence leaves it with the same state: a context menu is pushed (by t_1) and popped (by t_2), followed by a push and pop of `HostEditorActivity`, without any modification of the remaining stack. String $(push\ a) \circ \delta(T)$ is $push\ a, push\ m, pop\ m, push\ a', pop\ a'$ where a and a' denote `HostListActivity` and `HostEditorActivity` respectively, and m is the context menu attached to the clicked element in the list of a. This string satisfies the property defined in Definition 1. ◁

As another example, consider $T = \langle t_1, t_2\rangle$ in which t_1 and t_2 correspond to e_3 and e_4 in Figure 2. The sequence captures the following scenario: first press the hardware MENU button and then click the `sort` options menu item to sort the list items of `HostListActivity`. In this case, $(push\ a)\circ\delta(T) = push\ a, push\ m, pop\ m$ which also meet the neutrality property of a transition sequence. As a final example, if T contains only $t_1 = [a,a']$ triggered by a screen rotation event [a,*rotate*], then $\delta(T) = pop\ a, push\ a'$ because a is destroyed and recreated as a', and string $(push\ a)\circ\delta(T) = push\ a, pop\ a, push\ a'$ satisfies the desired property.

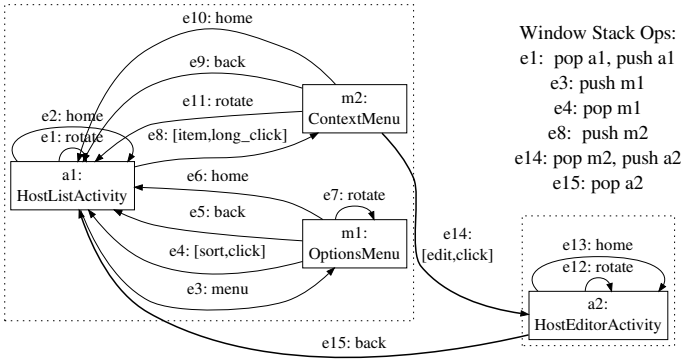

Figure 2: Window transition graph for the running example.

3. TEST GENERATION AND EXECUTION

The run-time behavior outlined earlier is first abstracted by a static model introduced in prior work. Next, paths in this model are used to define test coverage goals and the corresponding algorithms for test generation.

3.1 Window Transition Graph

For the rest of the paper we will use **Win**, **View**, etc. to denote sets of static abstractions rather than their run-time counterparts. We employ an approach from our prior work [24] in which activities, menus, dialogs, and views are modeled statically and their propagation through the code is tracked interprocedurally. Follow-up work [31, 30] defines the *window transition graph* (WTG), a static model with nodes $w \in \mathbf{Win}$ and edges $t = [w,w']$. Each transition t is annotated with trigger event $\epsilon(t)$, callback sequence $\sigma(t)$, and window stack changes $\delta(t)$. The public implementation of WTG construction [9] was used in our test generation.

The window stack changes $\delta(t)$ (i.e., window push/pop operations) are derived based on the window open/close operations performed by callback invocation $[c_i,o_i] \in \sigma(t)$. Control-flow analysis of c_i and its transitive callees is combined with a constant propagation data-flow analysis (based on the calling context o_i) to eliminate infeasible control-flow paths. This approach has been shown to produce precise static control-flow modeling [30].

The notion of a run-time neutral sequence can be naturally mapped to the notion of a *WTG neutral cycle*, based on the matching of push/pop window stack operations described in Definition 1. Since two run-time instances a and a' of the same activity class will be mapped to the same WTG node, the static abstraction is a WTG cycle. In addition, we impose the constraint that a WTG neutral cycle does not contain repeated edges, in order to disallow subcycles. Note that repeated nodes in the cycle are possible and desirable. For example, if one wanted to consider repeated executions of two widget events $e_1 = [v_1,k_1]$ and $e_2 = [v_2,k_2]$ such that the window stack does not change, the necessary WTG neutral cycle would be $a \xrightarrow{e_1} a \xrightarrow{e_2} a$.

The WTG for the running example is shown in Figure 2. This figure illustrates the window transitions and the event that triggers each transition. The window stack operations for a few of the transitions are also shown in the figure. The clusters of `HostListActivity` and `HostEditorActivity` are the nodes within the two dotted rectangles, respectively. Consider the WTG cycle $a_1 \xrightarrow{e_8} m_2 \xrightarrow{e_{14}} a_2 \xrightarrow{e_{15}} a_1$. This cycle shows a static abstraction of the leaky neutral transition sequence that is discussed at the end of Section 2.2. Another example of a neutral cycle is $a_1 \xrightarrow{e_3} m_1 \xrightarrow{e_4} a_1$.

Algorithm 1: GenerateIntraClusterNeutralCycles

```
1  foreach activity a ∈ Win do
2      path ← ⟨⟩
3      stack ← ⟨a⟩
4      Traverse(a, a, path, stack)
5  procedure Traverse(a, w, path, stack)
6      if path.length > k then
7          return
8      if path.length ≠ 0 ∧ path.endnode = a ∧ stack = ⟨a⟩ then
9          record path
10     foreach edge t = [w,w'] such that t ∉ path do
11         if CanAppend(a, t, path, stack) then
12             DoAppend(t, path, stack)
13             Traverse(a, w', path, stack)
14             UndoAppend(t, path, stack)
```

3.2 Intra- and Inter-Cluster Test Sequences

A common source of leaks is the mismanagement of resources during the lifecycle of an activity [13]. Thus, a natural target for test generation are repeated executions of neutral transition sequences that involve an activity a and all menus/dialogs that perform simple tasks on behalf of a—that is, only windows in $cluster(a)$. An *intra-cluster test sequence* is a WTG neutral cycle $T = \langle t_1, t_2, \ldots, t_n \rangle$ starting from node a such that each node in T belongs to $cluster(a)$. In Figure 2, an example of an intra-cluster neutral cycles is $a_1 \xrightarrow{e_3} m_1 \xrightarrow{e_4} a_1$ which corresponds to the sorting of list items as described in Section 2.2.

There are also examples of leaks that involve the interaction of two clusters: for example, there is a leak of binder objects when the user navigates from `HostEditorActivity` back to `HostListActivity`. Given two activities a and a', an *inter-cluster test sequence for a and a'* is a WTG neutral cycle $T = T_1 \circ T_2 \circ T_3$ starting at a and consisting of three components. First, the last node from T_1 is a' and the remaining nodes from T_1 are in $cluster(a)$. Similarly, all nodes from T_2 are in $cluster(a')$ except for the last one which is in $cluster(a)$. Finally, T_3 is entirely in $cluster(a)$. We also impose the restriction that each T_i does not contain repeated nodes. Since repetition of nodes is allowed in intra-cluster test sequences (as discussed earlier), such behaviors are not considered in the inter-cluster sequences. In Figure 2, an example of an inter-cluster neutral cycle is $a_1 \xrightarrow{e_8} m_2 \xrightarrow{e_{14}} a_2 \xrightarrow{e_{15}} a_1$.

Note that inter-cluster sequences could be generalized to involve more than two clusters. However, the number of test cases could become impractically large. Our experimental resuls indicate that cycles involving two clusters effectively expose leaking behaviors.

3.3 Test Generation

Our test generator is built on top of the publicly-available static analyses from [9, 24, 31]. Given the WTG, clustering is done as expected: for each activity a, a traversal of the WTG is peformed to find all menus and dialogs reachable from a without going through another activity. Set $cluster(a)$ contains a and all reached nodes.

For each activity $a \in \mathbf{Win}$, we perform a depth-first

traversal to explore intra-cluster neutral cycles. Algorithm 1 shows the detials of this traversal. The current WTG path and its corresponding window stack are maintained in *path* and *stack* respectively during the traversal. Any non-empty path, whose length is restricted by some analysis parameter k, ending with a that leaves the stack in the initial state is recorded for later processing to generate test cases.

During the depth-first traversal, helper function CanAppend filters out edges t that leave $cluster(a)$ and then determines whether current path could be appended with t via examining the validity of the window stack push/pop sequence by applying $\delta(t)$ to *stack*. Helper function DoAppend appends *path* with t and modifies *stack* according to $\delta(t)$ if CanAppend returns true. UndoAppend removes t from *path* and "undo" all the changes made to *stack*.

The construction of inter-cluster neutral cycles is conceptually similar, except that helper function CanAppend allows one edge of p to cross into another cluster, followed by a corresponding return edge.

3.4 Test Execution

The test cases are implemented in the Robotium testing framework [23]. For each transition t in a cycle T, event $\epsilon(t)$ is mapped to a corresponding Robotium API call. Some events may require additional manual inputs from the tester—e.g., to decide which item to click in a list, or which string to enter in a text field. Each test case is parameterized by a run-time parameter specifying the number of repetitions. Test case execution is similar to earlier work [28]: several resources are monitored and decisions are made based on these observations. We monitor memory usage and the number of threads/binders. Native memory is used by native code that is accessible via the Java Native Interface (JNI). Java heap memory is the memory space to store Java objects that is automatically managed by the Dalvik virtual machine. Binders and threads are used for inter-process communication and time-consuming tasks. Measurements for these resources can easily be collected via the Android Debug Bridge (ADB) and do not require any modifications of system code.

A test case is stopped early if it does not exhibit a pattern of growth. We employ the approach from [28]: a relatvely small number of repetitions (30 in our setup) of the neutral cycle is performed first, and linear regression on the measurements is used to compute rates of growth. A test case is stopped if the growth does not exceed a certain threshold. Around 90% of the test cases are stopped at this stage. The remaining test cases are allowed to continue running until either a crash is observed, or a limit on the number of repetitions (500 in our setup) is reached, after which the growth rate is analyzed as before to decide whether to report a leak.

To eliminate the effects of previous tests, the emulator is restarted every time before the execution of a test case. This takes about 1 minute. Test runs that stop early form the bulk of test execution time and one such run typically takes around 1 minute, excluding the emulator startup time. For the experiments described in the next section, we tested 8 applications for which the total number of generated test cases was around 50 thousand. By running test cases in parallel on several machines, we completed all experiments in around 2 weeks. It is important to note that our goal was not to reduce the running time of tests, but rather to perform comprehensive evaluation of the leak-detection capabilities of the proposed test generation. In future work we plan to investigate how to reduce this cost by adaptively choosing the number of neutral cycle repetitions, by focused run-time monitoring and amplification [7], and by using static or dynamic analysis to prioritize test cases.

4. EXPERIMENTAL EVALUATION

We evaluated the proposed testing approach on 8 open-source applications analyzed in our prior work [28]. While that earlier approach needs manual control-flow modeling and specification of neutral cycles, our current approach employs static analysis to generate the test cases automatically. The main goal of the evaluation was to answer the following research question:

> *RQ: Are the automatically-generated tests as effective as the manually-constructed ones in terms of detection of leak defects?*

The applications and the number of WTG nodes and edges are shown in the first three columns of Table 1. The large number of edges for `fbreader` is caused by a known precision limitation of prior static analyses [24, 30]. Column "Clusters" shows the number of clusters in the WTGs. The remaining columns show the numbers of generated test cases for different values of the path-length limit k (line 6 in Algorithm 1). Clearly, the number of test cases grows significantly with larger values of k, which raises interesting questions about test case prioritization, perhaps with the use of static or dynamic analysis of callbacks that are invoked in response to GUI events. For the rest of the experiments, we used $k=3$ as the parameter for test generation, except for `fbreader` for which we used $k=2$ due to the large WTG size. Test generation time is typically very low: for all but one program, it takes less than 1 second to generate all neutral cycles and the corresponding test cases (for `fbreader` this time is around 4 minutes). All test cases generated in this manner were executed as described in Section 3.4.

Table 2 compares the defects reported by our tests and compares them with the defects detected by the manual test cases from [28]. Here we measure unique leak defects and not the number of test cases, since it is possible for several test cases to fail due to a single defect. Column "Known" contains the number of defects already known from [28]. Columns "Intra" and "Inter" shows how many defects were reported by intra- and inter-cluster test cases, generated as described in Section 3. It can be seen that both test generation strategies are effective, and appear to be complementary.

Column $|K-D|$ is the number of known defects that were *not* reported by our tests. There are 2 such defects. In one case, limitations of the static analysis for WTG construction misses some window transitions, and the corresponding test cases are not generated. Further work on static analysis for Android can resolve this problem. The seconds case involves a neutral sequence of length which exceeds the path length limit. There is no easy way to resolve this issue: it is not feasible to test all long event sequences, and additional analyses or application-specific knowledge will likely be needed to select test cases that drive such behaviors.

The last column shows that 2 leak defects discovered in our experiments were new and were not discovered by the earlier test cases from [28]. One of these defects is a native memory leak in `connectbot`. It is caused by the incorrect implementation of the digest function in the framework. The

Application	WTG		Clusters	$k=2$		$k=3$		$k=4$		$k=5$	
	Nodes	Edges		Intra	Inter	Intra	Inter	Intra	Inter	Intra	Inter
apv	13	84	5	174	3	1557	13	14604	13	133619	13
astrid	53	374	19	733	23	8582	304	120628	864	1718582	1560
connectbot	34	192	11	305	7	1795	13	10401	13	56593	13
fbreader	40	26505	23	7689	2696	N/A	N/A	N/A	N/A	N/A	N/A
k9	30	356	18	1228	62	15142	128	209197	154	3012195	154
keepassdroid	29	308	15	842	58	10238	190	136902	302	1838782	386
vlc	19	70	10	307	5	2211	7	17763	7	144827	7
vudroid	7	47	4	42	9	142	17	438	19	1182	19

Table 1: Analyzed applications, number of clusters, and test cases.

Application	Known	Detected		$\lvert K-D\rvert$	$\lvert D-K\rvert$
		Intra	Inter		
apv	1	0	1	0	0
astrid	1	0	1	0	0
connectbot	3	3	1	0	1
fbreader	2	1	1	0	0
k9	4	3	0	1	0
keepassdroid	4	0	3	1	0
vlc	2	1	1	0	0
vudroid	1	0	2	0	1

Table 2: Resource leak defects: known and detected.

application leaks roughly 100 KB every time the password of a pubkey is changed. To ensure security, a password is digested for 1000 times by calling `MessageDigest.digest`. However, the byte array in this method is not fully released in native memory when it gets reset after the digest. This defect is fixed in Android Ice Cream Sandwich by upgrading the underlying library.

The other defect is a segmentation fault in `vudroid`, raised when the user tries to repeatedly open PDF files in a short period of time. This leak is also in native memory. It can be triggered by the neutral sequence of opening a PDF file and then going back to the previous activity. The application uninterruptedly renders a whole page of a PDF file and stores it in native memory, yet it does not release the memory in time after the file is closed.

Summary. These results show that, for the analyzed applications, the leak-detection effectiveness of the proposed test generation is comparable to the much more labor-intensive and error-prone manual approach from our prior work. Of course, the small number of experimental subjects poses a threat to the external validity (i.e., generalizability) of this conclusion. Still, we believe that this study presents promising initial evidence that automated testing for resource leaks in Android applications is feasible and effective.

5. RELATED WORK

Test generation for Android. A summary of many existing testing approaches for Android is presented in [6]. A few representative examples are discussed below. The standard Monkey tool [12] uses a simple random strategy for generating UI events. Android GUI Ripper [2] generates a dynamically built GUI model. MobiGUITAR [1] utilizes an enhanced version of AndroidRipper and then applies test adequacy criteria to it in order to generate test cases. It would be interesting to see whether this purely-dynamic approach (or some hybrid static-dynamic version of it) can be combined with our neutral-cycle-based testing for leaks. ORBIT [32] also uses a dynamic exploration strategy but combined with static code analysis to determine which UI events are relevant for a specific activity. The A^{3}E GUI exploration tool [4] employs two strategies: purely-dynamic depth-first exploration and targeted exploration based on a control-flow model from a static taint-like analysis. ACTEve [3] is a concolic testing tool which symbolically tracks events from their generation to their handling. Jensen et al. [17] use symbolic analysis to create event handler summaries and to build event sequences using the summaries and a UI model. Other examples of testing tools for Android include Dynodroid [19], SwiftHand [25], EvoDroid [20], PUMA [15], and Droidmate [16].

Test amplification for Android. Amplification can be used to detect potential problems during testing. Zhang and Elbaum [33] apply test amplification techniques for exception-handling code, which is necessary due to unreliable environment factors such as network connectivity. Their approach amplifies existing test cases by injecting exceptions. Our prior work [29] amplified potentially-expensive operations that could affect the responsiveness of the UI thread and lead to application-not-responding errors. Fang et al. [7] develop a tool to quickly find memory-related performance problems in managed languages by amplifying the size of allocated objects. It may be possible to enhance our approach with memory object amplification to reach crashes faster; this issue is orthogonal to the problem of test generation.

Leak analysis for Android. Pathak et al. defined a static analysis to find defects in which code paths leak energy-related resources [21]. Another static analysis [14] aims to examine Android event handlers for the usage of resource-related APIs. Neither of these approaches models the full generality of GUI control flow. Our WTG model is a state-of-the-art static representation of Android GUI, and the only one that models the state of the window stack. Other leak analyses use purely-dynamic techniques. GreenDroid [18] uses Java PathFinder to trigger various GUI event sequences based on a manual model of GUI structure and behavior, in order to observe dynamically leaks related to battery drain. Another energy bug detection tool [5] is also based on dynamic analysis of resource leaks. A modified version of Dynodroid [19] is employed to generate a model of the GUI, and the model is then used to trigger sequences of GUI events. Both approaches aim to detect energy-leak behaviors expressed as misuse of acquire-release API calls. As recent work shows [6], 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. Our test generation, based on a comprehensive

static model that identifies neutral event sequences, provides an alternative to these approaches.

6. CONCLUSIONS AND FUTURE WORK

Using static control-flow analysis, it is possible to automatically create comprehensive tests to cover various repetitive behaviors in Android GUIs. Our initial study strongly suggests that significant leak detection can be achieved automatically, at a level comparable with test generation that involves manual effort. Future work needs to develop techniques for test prioritization using static or dynamic analysis, as well as refinements of test execution and monitoring (e.g., by using amplification techniques in the spirit of [7]).

7. ACKNOWLEDGMENTS

We thank the AST 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.

8. REFERENCES

[1] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, pages 53–59, 2015.

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

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

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

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

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

[7] L. Fang, L. Dou, and G. Xu. Perfblower: Quickly detecting memory-related performance problems via amplification. In *ECOOP*, pages 296–320, 2015.

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

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

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

[11] Google. Managing your app's memory. `developer.android.com/training/articles/memory.html`.

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

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

[14] 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.

[15] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *MobiSys*, pages 204–217, 2014.

[16] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller. BOXMATE. `boxmate.org`.

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

[18] 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.

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

[20] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: Segmented evolutionary testing of Android apps. In *FSE*, pages 599–609, 2014.

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

[22] T. Reps. Program analysis via graph reachability. *IST*, 40(11-12):701–726, 1998.

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

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

[25] C. Wontae, N. George, and S. Koushik. Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA*, pages 623–640, 2013.

[26] M. Xia, W. He, X. Liu, and J. Liu. Why application errors drain battery easily? In *HotPower*, pages 2:1–2:5, 2013.

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

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

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

[30] 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.

[31] 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.

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

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