Systematic Testing for Resource Leaks in Android Applications

Dacong Yan   Shengqian Yang   Atanas Rountev
Ohio State University

Abstract—**The use of mobile devices and the complexity of their software continue to grow rapidly. This growth presents significant challenges for software correctness and performance. In addition to traditional defects, a key consideration are defects related to the *limited resources* available on these devices. Resource leaks in an application, due to improper management of resources, can lead to slowdowns, crashes, and negative user experience. Despite a large body of existing work on leak detection, testing for resource leaks remains a challenging problem. We propose a novel and comprehensive approach for systematic testing for resource leaks in Android software. Similar to existing testing techniques, the approach is based on a GUI model, but is focused specifically on coverage criteria aimed at resource leak defects. These criteria are based on *neutral cycles*: sequences of GUI events that should have a "neutral" effect and should not lead to increases in resource usage. Several important categories of neutral cycles are considered in the proposed test coverage criteria. Experimental evaluation and case studies were performed on eight Android applications. The approach exposed 18 resource leak defects, 12 of which were previously unknown. These results provide motivation for future work on analysis, testing, and prevention of resource leaks in Android software.**

I. Introduction

Android devices currently lead the smartphone marketplace in the United States [1] and similar trends can be seen in other countries. Android also has significant presence in one of the fastest-growing segments of the computing landscape: tablets (e.g., Google Nexus 7/10, Samsung Galaxy Tab/Note) and media-delivery devices (e.g., Amazon Kindle Fire, Barnes & Noble Nook HD). The widespread use of these mobile devices poses great demands on software quality. However, meeting these demands is very challenging. Both the software platforms and the accumulated developer expertise are immature compared to older areas of computing (e.g., desktop applications and server software). The available research expertise and automated tool support are also very limited. It is critical for software engineering researchers to contribute both foundational approaches and practical tools toward higher-quality software for mobile devices.

A. Resource Leaks in Android Applications

The features of Android devices and the complexity of their software continue to grow rapidly. This growth presents significant challenges for software correctness and performance. In addition to traditional defects, a key consideration are defects related to the *limited resources* available on these devices. One such resource is the memory. In Android's Dalvik Java virtual machine (VM) the available heap memory typically ranges from 16 MB to 64 MB. In contrast, in a desktop/laptop VM there are many hundreds of MB available in the heap. Examples of other limited resources include threads, binders (used for Android's inter-process communication), file handles,

and bitmaps. An application that consumes too many resources can lead to slowdowns, crashes, and negative user experience.

Resource management is challenging and developers are made aware of this problem in basic Android training materials [2] and through best-practice guidelines (e.g., [3]), with the goal of avoiding common pitfalls related to resource usage. A typical example of such a problem is a *resource leak*, where the application does not release some resource appropriately.

Examples. We studied a version of `ConnectBot` [4], an SSH client with more than a million installs according to the Google app store. The code contains a leak: when the application repeatedly connects with a server and subsequently disconnects from it, bitmaps are leaked, which eventually leads to a crash. As another example, we studied a version of the `APV` PDF viewer [5] (which also has more than a million installs) and discovered a leak, occurring when a PDF file is opened and then later the BACK button is pressed to close the file. In our experience, leak defects are related to diverse categories of events such as screen rotation, switching between applications, pressing the BACK button, opening and closing of files, and database accesses. If application users observe crashes and slowdowns due to such leaks, they may uninstall the application and submit a negative review/rating in the application marketplace.

Challenges. Even though resource leaks can significantly affect software reliability and user experience, *there does not exist a comprehensive and principled approach for testing for such leaks*. The large body of work on dynamic analysis of memory leaks (e.g., [6]–[13]) has the following purposes: (1) observe run-time symptoms that indicate a potential leak, and (2) provide information to diagnose the root cause of the defect (e.g., by identifying fast-growing object subgraphs on the heap). However, all these approaches fail to address one crucial question: how can we generate the test data that triggers the leaking behavior? Answering this question for arbitrary applications is difficult, because leaks may be related to a wide variety of program functionality. However, as discussed later, a key insight of our approach is that leaks in Android applications *often follow a small number of behavioral patterns*, which makes it possible to perform systematic, targeted, and effective generation of test cases to expose such leaks.

Each Android application is centered around a graphical user interface (GUI), defined and managed through standard mechanisms provided by the Android platform. Some leak patterns are directly related to aspects of these mechanisms—for example, the management of the lifetime for an *activity* [14], which is an application component that interacts with the user. Such leaks cannot be exposed through unit testing because of the complex execution context managed by the platform (e.g., lifetime and internal state of GUI widgets, persistent state,

etc.), as well as the complicated interactions due to callbacks from the platform to the application. It is essential to develop a system-level GUI-centric approach for testing for Android leaks, with sequences of GUI events being triggered to exhibit the leak symptoms. At present, no such approach exists.

B. Our Proposal

We propose a *novel and comprehensive approach for testing for resource leaks in Android software*. This leak testing is similar to traditional GUI-model-based testing. Finite state machines and other related GUI models have been used in a number of testing techniques (e.g., [15]–[20]), including recent work on testing for Android software [21]–[25]. Given a GUI model, test cases can be generated based on various coverage criteria (e.g., [16]). As with these existing approaches, we consider GUI-model-based testing, but focused specifically on coverage criteria aimed at resource leaks. We define the approach based on a GUI model in which nodes represent Android activities and edges correspond to user-generated and framework-generated events. The same approach can be used with other GUI models for Android (e.g., event-flow graphs [18], [26]) in which paths in the model correspond to event sequences.

The proposed coverage criteria are based on the notion of *neutral cycles*. A neutral cycle is a sequence of GUI events that should have a "neutral" effect—that is, it should not lead to increases in resource usage. Such sequences correspond to certain cycles in the GUI model. Through multiple traversals of a neutral cycle (e.g., rotating the screen multiple times; repeated switching between apps; repeatedly opening and closing a file), a test case aims to expose leaks. This approach directly targets several common leak patterns in Android applications, and successfully uncovers 18 resource leak defects in a set of eight open-source Android applications used in our studies.

Contributions. The contributions of this work are:

- *Test coverage criteria:* We define several test coverage criteria based on different categories of neutral cycles in the GUI model. This approach is informed by knowledge of typical causes of resource leaks in Android software.
- *Test generation and execution:* We describe LEAKDROID, a tool that generates test cases to trigger repeated execution of neutral cycles. When the test cases are executed, resource usage is monitored for suspicious behaviors.
- *Evaluation:* We evaluate the approach on several Android applications. The evaluation demonstrates that the proposed test generation effectively uncovers a variety of resource leaks.
- *Case studies:* We present case studies of leak defects exposed by the approach. This provides insights into the root causes of these leaks, which may be useful for future work on testing and debugging of Android software.

These contributions are in the emerging and important area of software testing for mobile devices. The proposed testing approach adds to a growing body of research on improving the reliability and performance of Android applications. The experimental evaluation and case studies contribute to better understanding of certain classes of defects in such applications, and highlight open problems for future investigations.
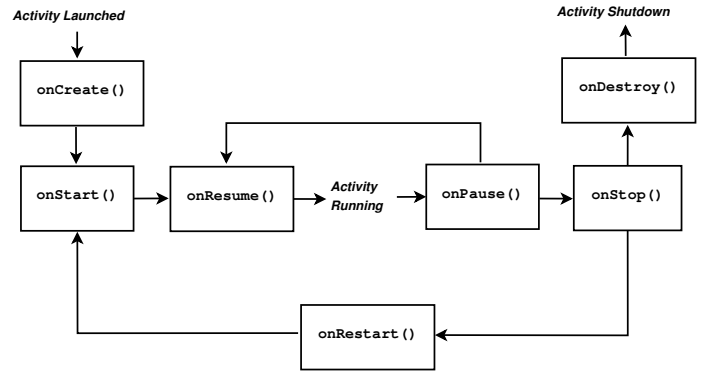

Fig. 1. Activity lifecycle.

II. BACKGROUND

A. Android Activities

An Android *activity* is an application component that manages a hierarchy of GUI widgets and uses them to interact with the user. An activity has a well-defined lifecycle, and developers can define callback methods to handle different stages of this lifecycle (Figure 1). When an activity is started, `onCreate` is called on it by the Android runtime. The activity becomes ready to terminate after `onDestroy` is called on it. The loop defined by `onStart` and `onStop` is the *visible* lifetime. Between calls to these two callback methods, the activity is visible to users. Finally, the innermost loop `onResume`/`onPause` defines the *foreground* lifetime, in which the activity is on the foreground and can interact with the user. A resource leak can be introduced if a certain resource is allocated at the beginning of a lifetime (e.g., in `onCreate`) but not reclaimed at the end (e.g., in `onDestroy`). Thus, one desirable property of a test generation strategy is to cover these three pairs of lifecycle callback methods, especially because prior studies of Android applications [27] indicate that defects are often caused by incorrect handling of the activity lifecycle. An application usually has several activities, and transitions between them are triggered through GUI events. When an application is launched, a start activity is first displayed.

Example. Figure 2(a) shows `ChooseFileActivity` in the `APV` PDF viewer application [5], displayed when the application is launched. The activity shows a list of files and folders. A PDF file can be selected by tapping on the corresponding list item, and the file is displayed in `OpenFileActivity` as shown in Figure 2(b). These two activities correspond to two different states of the application; each has its own visible GUI elements and allowed GUI events. The reverse transition occurs through the hardware BACK button. This transition closes the file and returns to the previous screen. The sequence of operations that opens a file and then closes it is expected to have a "neutral" effect on resource usage, and is an example of a neutral cycle. Repeated execution of this cycle normally should not lead to a sustained pattern of resource usage growth.
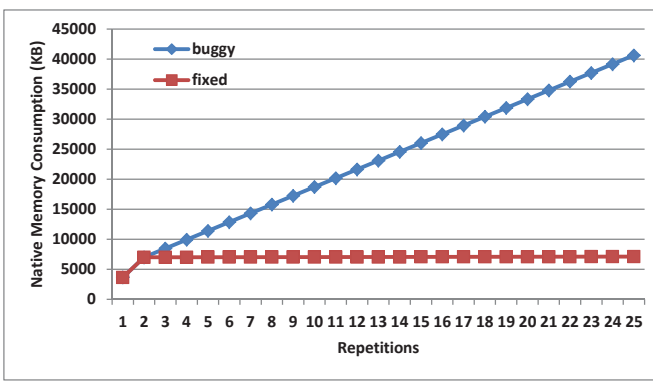
When executing an automated test case that repeatedly exercises these two transitions (selecting a file and then pressing the BACK button), we observed that the native memory usage increases significantly and ultimately leads to a crash. After examining the application code, we determined that
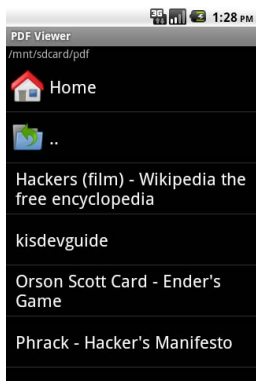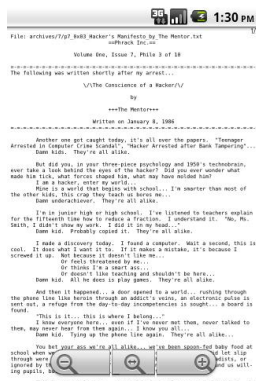
Fig. 2. APV application: (a) `ChooseFileActivity` lists files and folders. (b) `OpenFileActivity` displays the selected PDF file. (c) Native memory usage before and after fixing the leak.

certain amount of native memory is allocated during the initialization of `OpenFileActivity` and freed when the PDF file is closed, via a call to a native method `freeMemory`. However, `freeMemory` does not free all allocated memory, which results in a memory leak. In fact, in a later version of the application, the developers checked in a fix for this issue. The native memory consumption before and after this fix are shown in Figure 2(c); the x-axis shows the number of repetitions of the neutral cycle.

B. Leak Testing with a GUI Model

Following a large body of work on GUI-model-based testing [15]–[25], the starting point of our approach is a model of the Android application's GUI. To focus the presentation, we discuss one particular kind of model. However, the notion of neutral cycles and the coverage criteria based on them should be easily applicable to other GUI models (e.g., [18], [26]), where there is a natural correspondence between paths in the model and sequences of events. A partial GUI model for APV is shown in Figure 3. The figure shows only a subset of GUI states and transitions, as needed for explanation purposes.

The models we discuss are directed graphs, with one node per activity, and with edges representing transitions triggered by GUI events. The set of nodes is defined by the set of application classes that subclass (directly or transitively) class `android.app.Activity`: each such class is a node in the model. In addition to traditional events, the model should capture Android-specific events. For example, a user can press the hardware MENU button and then select a menu item from a list specific to the current activity. In Figure 3, edges labeled with MENU: represent such events; for example, MENU:About corresponds to choosing the "About" menu item. As another example, the hardware BACK button can be used to destroy the current activity and to transition to another one. (Although the programmer can choose to override this BACK button behavior with application-specific logic.) In addition to such application-specific events, several important GUI events are defined by the platform and not by the application:

ROTATE events. When the user rotates the screen, the current activity is recreated with a different orientation. In the model this event is represented by a self-transition labeled
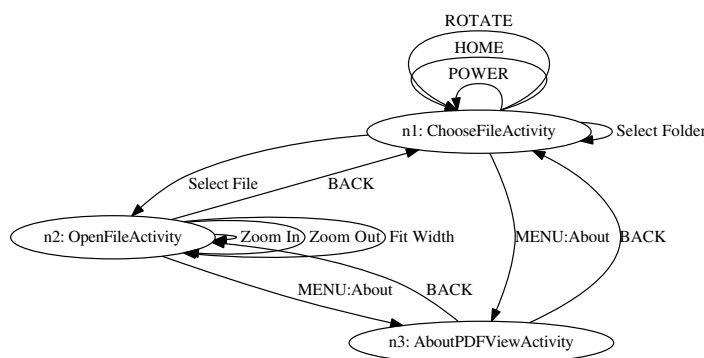

Fig. 3. A subset of the GUI model for APV.

with ROTATE. A rotation event is important for testing because it covers the `onCreate`/`onDestroy` pair in the activity lifecycle from Figure 1. It is well known that repeated execution of this pair of methods can leak activity objects (instances of `android.app.Activity`), GUI widget objects (instances of `android.view.View`), visual resources (instances of `android.graphics.drawable.Drawable`) such as bitmaps, and other categories of resources [2], [3]. To simplify Figure 3, only the ROTATE edge for n_1 is shown; both n_2 and n_3 have similar edges.

HOME events. When the user presses the hardware HOME button, the application is hidden. The launcher, a special application to allow the user to launch any application, is then brought to the foreground. For testing purposes, we are interested in the scenario where the original application is immediately selected to be reactivated. Edge HOME in Figure 3 represents pressing HOME and then going back to the same application. (A similar self-edge exists for each other node in the model.) Another situation with behavior equivalent to a HOME event is when the user receives a phone call while the application is active; once the phone call is completed, the application is reactivated. A HOME transition corresponds to the `onStart`/`onStop` loop in Figure 1 and could be considered for coverage during testing.

POWER events. The hardware POWER button puts the device in a low-power state. In this case, `onPause` is called on the current activity. When the button is pressed again

and the screen is unlocked, the activity becomes active and its `onResume` method is called. Edge POWER in Figure 3 represents this sequence of operations. The same behavior and callbacks are observed in other scenarios unrelated to power usage—e.g., when an activity is partially blocked by a popup dialog. A testing strategy could consider coverage of POWER transitions.

Sensor events. The platform can generate other events due to user actions. For example, an accelerometer can trigger events because of shaking or tilting motions. More generally, acceleration forces and rotational forces can be sensed by accelerometers, gravity sensors, gyroscopes, and rotational vector sensors [28]. These sensor events are GUI events triggered by the user, and they can activate interesting behaviors. Our current approach does not include these events, but can be easily extended to consider them as well.

C. Obtaining GUI Models

Various reverse-engineering techniques (e.g., [17], [20], [24], [25], [29]) can be used to automatically construct GUI models. A recent example is AndroidRipper [22], [23], [30], a tool to perform GUI reverse engineering for Android applications. Its implementation uses the Robotium testing framework [31] to systematically explore the GUI. At each GUI state, the tool examines the run-time GUI widgets and the events that can be fired upon them. The models produced by the tool are very detailed. For example, a MENU transition is represented by two edges, one for pressing the hardware MENU button and another for choosing a menu item (e.g., "About"). As another illustration of this level of detail, the same activity may be represented by many states in the model. For example, there are many possible lists of files/folders that can be displayed by activity `ChooseFileActivity` shown in Figure 2(a), by following the "parent folder" list item (labeled with ".." in the figure), or another list item representing a subfolder. Each such file/folder list would be represented by a different state, resulting in a very large model.

To reduce model size and the number of generated test cases, we chose to use an abstracted model with one-to-one correspondence between activities and model states. For our experiments these models were created manually after examining the output of AndroidRipper and the source code of the application. We also added HOME and POWER transitions, which were not captured by AndroidRipper. It was an intentional decision *not* to focus on fully automating the model construction, but instead focus on evaluating the model-based coverage criteria and showing that they are indeed useful for exposing leak defects.

III. GENERATION AND EXECUTION OF TEST CASES

The testing approach is based on a set of *test coverage criteria*. Each criterion is aimed at a particular category of neutral cycles in the model of the application's GUI. Note that we expect this kind of leak testing to be performed after—and be complementary to—traditional functional testing during which high block/branch coverage is achieved. Thus, we focus specifically on coverage of repeated behavior that may be related to leaks.

A. Test Coverage Criteria

To illustrate a coverage criterion, consider the ROTATE transition shown in Figure 3. In general, for each state n_i in the model, there is a self-transition representing a ROTATE event. We can define the following coverage criterion: for each state n_i, execute at least one test case that corresponds to a path $(s, \ldots, n_i, n_i, \ldots, n_i)$. Here s is the start state, prefix $(s, \ldots, n_i)$ represents a cycle-free path, and suffix $(n_i, n_i, \ldots, n_i)$ contains only ROTATE transitions. This suffix corresponds to k repetitions of the neutral cycle $n_i \rightarrow n_i$. The motivation for this coverage is clear: resource usage should not increase when the screen is rotated repeatedly [3], even for large k. Executions of this cycle will trigger repeated `onCreate`/`onDestroy` lifecycle callbacks (recall Figure 1). As mentioned earlier, resource leaks often occur because of defects related to lifecycle management. We have seen a number of examples of this pattern in our studies.

Application-independent cycles. One category of cycles to be covered are those defined by ROTATE, HOME, and POWER events—i.e., events defined by the platform, not by the application. An example of a ROTATE-based coverage was given above. Similar coverage can be defined for HOME cycles (to trigger repeated `onStart`/`onStop`) and POWER cycles (for repeated `onPause`/`onResume`). Note that even though repeated ROTATE events also result in repeated start/stop and pause/resume, they do not necessarily expose leaks related to stopping or pausing an activity: because ROTATE destroys the activity, it may release resources that are leaked by `onStop` or `onPause`. We have observed this situation in our studies.

Cycles with BACK transitions. The coverage criteria described above target only the activity that is currently interacting with the user. Cycles involving the hardware BACK button involve multiple activities, and present another target for coverage. For each BACK transition $n_i \rightarrow n_j$, we can execute a path $(s, \ldots, (n_j, \ldots, n_i)^k, n_j)$. Here the k transitions from n_i to n_j are done with the BACK button, and the shortest path from n_j to n_i is taken each time to reach the BACK edge. In our experience, cycle $(n_j, \ldots, n_i, n_j)$ is invariably a neutral cycle: resource usage growth over multiple repetitions is unexpected and suspicious. Coverage of cycles involving BACK edges may expose leaks that depend on the interplay among several activities. For example, we have observed cases where coverage of single-activity cycles (e.g., ROTATE cycles) does not expose a leak, but coverage of cycles with BACK transitions triggers the leaking behavior.

Application-specific neutral operations. We also consider cycles involving pairs of operations that "neutralize" each other. For example, node n_2 in Figure 3 has two self-transitions "zoom in" and "zoom out", triggered by two of the buttons shown at the bottom of Figure 2(b). The zooming-in operation, followed by the zooming-out one, should have a neutral effect, and a neutral cycle can be defined with these two operations. Other examples include connecting to/disconnecting from a server, opening/closing a file, adding an email account and then deleting it, etc. In addition, a single operation that only refreshes the GUI state of an activity (e.g., refreshing a list of email messages) should have neutral effect on resource usage.

Test case context. For a neutral cycle $(n_i, \ldots, n_i)$, any executable test case must contain a prefix path $(s, \ldots, n_i)$

```
1  // @PreCondition
2  //    A PDF file at position 3 of list
3  void test_n3_BACK_n2() {
4    robotium.clickInList(3);  // n1 -> n2
5    // Cycle: n2 -> n3 -> n2 -> ...
6    for (int i = 0; i < k; i++) {
7      robotium.clickOnMenuItem("About");
8      robotium.goBack();
9    }
10 }
```

Fig. 4.   An example of a generated test case.

where s is the start state. How should this prefix be chosen? In our current approach, we choose the shortest path from s to n_i. However, *context-sensitive* variations of the coverage could also be defined, where different execution contexts for the neutral cycle (i.e., different prefix paths leading to n_i) need to be covered. Making such choices is very similar to defining different calling contexts for functions in code analysis and testing, and presents interesting opportunities for future work.

B. Test Generation and Execution

Given a GUI model and a coverage goal, test generation can be achieved by traversing paths in the model. We have developed LEAKDROID, a tool that implements this approach. In the generated test cases GUI events are triggered with the help of the Robotium testing framework [31]. A test case is shown in Figure 4. It corresponds to a path $(s = n_1, (n_2, n_3)^k, n_2)$ in the GUI model from Figure 3, and covers the BACK edge from n_3 to n_2. The start state is n_1. Line 4 makes an API call to select the third list item, assuming that the item represents a PDF file, and makes the transition to state n_2. The loop at lines 6–9 executes k repetitions of a neutral cycle that involves the BACK edge $n_3 \rightarrow n_2$. The call at line 7 selects a menu item, and the call at line 8 presses the BACK button. The API calls for GUI events are generated automatically by LEAKDROID based on the given model and the coverage goal. The tool input also includes information about application-specific pairs of operations with neutral effects (e.g., open/close) and single neutral operations (e.g., refresh). Data-specific elements (e.g., choosing the third list item at line 4) are subsequently provided by the tester. We found that the manual effort for this is trivial—once the Robotium calls are generated automatically, test setup (e.g., setting up an SSH host name at a specific position in the host list, or a file name at a certain position in the file list) is very easy.

During test case execution, various resources can be monitored. Currently we collect the following measurements.

Java heap memory. This is the memory space used to store Java objects. Existing memory leak detection techniques for Java typically focus on leaks in this memory space. The space is automatically managed by the garbage collector, so there can be leaks only when unused objects are unnecessarily referenced. Note that some resource leaks (e.g., leaking of database `Cursor` objects) also exhibit usage growth in this memory space.

Native memory. This memory space is used by native code, and is made accessible to Java code via JNI (Java Native Interface) calls. It requires explicit memory management by the developers as in programs written in non-garbage-collected languages such as C/C++, and thus could suffer from all well-known memory-related defects in those languages (e.g., dangling pointers, double-free errors). For example, the native `recycle` method of the `Bitmap` class has to be explicitly called to prevent leaking of native bitmap objects. This memory space is particularly important to monitor as many Android applications make heavy use of native code and thus native memory.

Binders. Binders provide an efficient inter-process communication mechanism in Android. In essence, a binder is the core component of a high-performance remote procedure call (RPC) mechanism directly supported by the underlying Linux kernel in the Android operating system. Usage of binders requires creation of global JNI references, and these references are made visible to the garbage collector. Unnecessarily keeping these references could lead to leaking of other potentially large Java objects. The global JNI references are deleted in native methods called by the finalizer of `android.os.Binder`, so the number of `Binder` instances is a good indicator of whether unnecessary JNI references are kept. There is likely to be an underlying software defect if this number grows significantly, and we collect measurements of it to identify binder leaks. Such leaks are distinguished from memory leaks because they are related to an Android-specific feature and behavior, which allows more precise diagnosis of the root problem.

Threads. Threads are usually created to perform time-consuming operations in a GUI application to maintain good responsiveness. For example, the e-book reader `VuDroid` [32] creates new threads to compute rendering data for requested files. A buggy implementation could hang thread execution, while new threads are being created. A sustained growth in the number of active threads in an application is an indication of software defects, and thus the proposed testing approach collects measurements of the number of active threads.

All of the discussed measurements can be easily collected via system services provided by the Android platform, and does not require any code changes or system modifications. To reduce the running time for test execution, we stop a test case early if it does not exhibit a pattern of growth. Various techniques can be used to decide whether a test case should be stopped. Currently we use a technique which monitors resource usage for 500 repetitions of the neutral cycle, performs linear regression on the measurements, and stops the test case if the rate of growth is below a certain threshold (e.g., less than 5% memory growth per hour). Although simple, this technique stops early the majority of test cases (76% in our experiments), allowing testing resources to be focused on a smaller set of test cases with non-trivial growth in resource usage. Each such "suspicious" test case is executed until it fails or until a predefined limit on the number of neutral cycle repetitions is reached. An interesting observation is that some non-failing test cases exhibit slow-leak behavior: there is a pattern of slow growth that may indicate an underlying defect. Our current reporting and evaluation focus only on failing test cases, in which a defect is clearly manifested; slow leaks will be investigated in future work.

C. Diagnosis of Failing Test Cases

When a test case fails, various techniques can be used to diagnose the root cause. For example, heap snapshots and

object reference graphs derived from them are available in a number of tools (e.g., [6]). Information derived from such graphs is often analyzed manually to understand memory usage and diagnose memory leaks in Android applications [33]. Various automated analyses of heap graphs have also been proposed (e.g., [8], [10]). Such analyses can potentially be extended to reflect the structure of the generated test cases. For example, a crashing test case that exhibits memory growth can be re-executed with a small number n of repetitions of the neutral cycle. As the test case is running, a heap snapshot is taken after each cycle repetition. After re-execution, n heap snapshots $H_1, H_2, \ldots, H_n$ are available, and $n-1$ heap differences $\Delta_i = H_{i+1} - H_i$ can be computed and analyzed. Our initial experience with manually applying this approach was very promising, and helped to identify the causes of all memory-growth test cases we observed. The diagnosis was performed with the help of the MAT memory analysis tool [6] (which is commonly used by Android developers [3]), followed by code inspection. An interesting question for future work is how to apply this approach to automated heap-differencing techniques (e.g., [8], [10]) and how to generalize it for analysis of native memory and resources other than memory.

IV. EVALUATION

We evaluated the proposed testing approach on eight open-source Android applications. The test cases were generated with our LEAKDROID tool. We debugged all failing test cases and identified the underlying defects. All experiments were performed in the standard Android emulator from the Android SDK. The experimental subjects, their GUI models, the test cases, the description of identified defects, and the source code of LEAKDROID are all publicly available at www.cse.ohio-state.edu/presto/software.

A. Study Subjects

We used search engines to establish a set of potential study subjects. The subjects were restricted to open-source Android applications; however, the proposed approach can be easily applied to applications without publicly accessible source code. Applications that were less popular (e.g., with only a few installs) or not well-maintained (e.g., applications without a bug database, with only a few commits) were excluded from consideration. For an initial set of candidate applications, we searched their bug databases and code commit log messages. Search terms such as "leaks" and "out of memory error" were used to identify application versions that may contain leak defects. During or after this process, we did *not* examine carefully the bug reports and code commits, in order to ensure that the test cases generated by our approach were not biased toward any particular existing faults.

Characteristics of the study subjects are shown in the first five columns of Table I. The number of application classes that subclass android.app.Activity is shown in column "Activities/States". Even for applications with only a few activities (e.g., APV), there could be several dozen other application classes to provide supporting functionality for the activities, which can lead to complicated run-time behavior. Each activity shown in "Activities/States" corresponds to a state in the GUI model. Column "Transitions" shows the number of edges in the model. This number does not include implicit application-independent self-transitions (that is, ROTATE, HOME, and POWER transitions). The applications represent a variety of domains: e-book/PDF readers (APV, FBReader, VuDroid), to-do list management tool (astrid), email client (K9), SSH client (ConnectBot), password manager (KeePassDroid), and multimedia player (VLC).

B. Experimental Methodology

For each application, test cases were generated (as described in Section III-B) to achieve complete coverage with respect to the test coverage criteria defined earlier. Since ROTATE/HOME/POWER transitions exist for each state in the GUI model, the test cases are guaranteed to cover each activity in the application. Next, all generated test cases were executed as described in Section III-B. During execution, usage measurements for various resources were collected for detection of growth patterns. When a test case fails, these measurements provide some initial clues as to what type of resource is leaking and what could be the underlying defect. For each failing test case, we investigated the application (using code inspection and a memory analysis tool) to determine the root cause of the failure and whether this cause was indeed related to resource leaks. Details of this investigation are presented in Section V.

On average the execution time for a failing test case is less than two hours, with the majority of test cases failing in less than an hour. These times are artificially inflated due to a particular deficiency of our initial implementation. When firing an event through a Robotium call (recall Figure 4), it is necessary to wait until the effects of the event are processed by the application and shown in the GUI, so that the next event can be fired on the updated GUI. In our current prototype implementation we automatically introduce a large delay after each event in the test case. It is an interesting open problem how to automatically fine-tune the durations of these delays in order to reduce test execution time; we plan to investigate this problem in the near future.

C. Detection of Leak Defects

The rest of Table I shows measurements to demonstrate the effectiveness of the coverage criteria in detecting leak-related defects. Shown in order are the number of generated test cases, the number of failing test cases due to memory leaks, the number of failing test cases due to thread leaks, the number of failing test cases due to binder leaks, and the number of unique leak defects exposed by these failing tests (and confirmed by us through investigation of the source code).

Column "Test Cases" shows how many test cases were generated based on the coverage criteria described earlier. The test cases tend to be relatively small: on average, the number of Robotium calls per test case (i.e., the number of events fired) was 4.04. As discussed in Section III-B, some of these test cases were filtered out early in their execution: we used a filtering approach to detect growth patterns and stop test cases that are not likely to cause sustained growth in resource usage. This approach was quite effective, and only 24% of the generated test cases needed to be executed further after the filtering step. Test cases that cover the POWER events are not included in these measurements, because we did not observe any resource usage growth related to these events.

Application	Version	Activities/ States	Transitions	Classes	Test Cases	Memory Leaks	Thread Leaks	Binder Leaks	Unique Defects
APV	r131	4	16	56	22	1	0	0	1
astrid	cb66457	11	27	481	40	3	0	0	1
ConnectBot	e63ffdd	9	27	301	32	3	0	10	3
FBReader	a53ed81	22	31	757	30	6	0	0	2
KeePassDroid	085f2d8	7	30	126	33	4	0	1	4
K9	v0.114	15	45	418	57	4	0	16	4
VLC	dd3d61f	8	22	176	32	4	0	0	2
VuDroid	r51	3	11	67	17	0	2	0	1

A test case is included in column "Memory Leaks" when a crash is caused by a memory leak that leads to an out-of-memory error. It can be a memory leak in either the managed heap or the native heap. These memory leaks could have various underlying reasons, and some of them are related to inappropriate management of resources—for example, leaking of bitmap objects, database cursors, and event listeners. Some of these resources have memory budgets (e.g., bitmaps). Exceeding the budget limit will immediately lead to an out-of-memory error, although there could still be sufficient memory space left in the heap. Thus, it is important to force immediate reclamation of such resources when they are no longer needed.

"Thread Leaks" refers to the test cases that have a large number (100 in our experiments) of simultaneously active threads. It is our experience that an application exhibiting such behavior is very likely to have a thread leak problem, and it is unnecessary to wait until it exceeds the system-wide limit on the number of threads, which is usually even larger (4096 for Android). In fact, such test cases could crash very quickly when the amount of memory reachable from each newly-spawned thread is substantial.

In the Android emulator, there is a system-wide limit on the maximum number of global JNI references. When an application exceeds this limit, the emulator crashes. Although this limit by default is not enabled on real Android devices, exceeding it is still an indication of software defects. As discussed earlier (Section III-B), usage of binders requires creation of global JNI references. So, there will be a crash when an application keeps creating new binder objects and maintains references to them. Column "Binder Leaks" counts the number of failing test cases that fall into this category.

The last column in the table shows the number of defects that are responsible for the failing test cases. Details on some of these problems are presented shortly. Among the 18 defects discovered, only 6 could be connected to existing bug reports and code commit logs; the remaining 12 were previously unknown. For the 6 defects related to existing reports/commits, this relationship was established (by examining bug reports and commit logs) only *after* the defects were uncovered by the generated test cases and confirmed by code examination. The 18 discovered defects were exposed using only a systematic model-based test generation approach, without any prior knowledge of their symptoms and root causes.

Note that it is impossible to ascertain how many leak defects actually exist in the studied applications. The vast majority of bug reports are vague and do not provide enough information to construct an actual test case to reproduce the
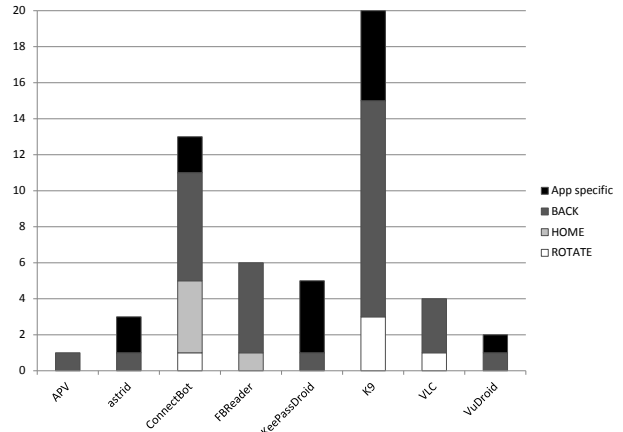

Fig. 5.   Failing test cases for each category of neutral cycles.

failure, or to identify its root cause. Similarly, code commits typically have short and uninformative commit messages, again without details on how to reproduce the incorrect behavior being fixed. Only after generating our model-based test cases and debugging the failing ones, we have been able to determine that some existing bug reports and code commits are referring to the same defects.

D. Defect Detection for Coverage Criteria

We have defined and used several coverage criteria to target various types of neutral cycles. To understand the leak detection capabilities of each kind of neutral cycle, we categorized failing test cases based on the type of cycles they exercise. Figure 5 provides a summary of this study. The chart shows the numbers of failing test cases that exercise a neutral cycle to cover ROTATE transitions, HOME transitions, BACK transitions, and application-specific operations (a single neutral operation or a pair of neutralizing operations). The last two categories of neutral cycles exhibit the best ability to uncover leak defects. These cycles often involve both resource allocation and reclamation (reclamation could be missing if the application has a leak defect). Application-independent cycles (i.e., cycles defined by ROTATE, HOME, and POWER transitions) have weaker leak detection capabilities. As mentioned earlier, test cases that exercise POWER transitions were excluded from the presented measurements because they do not exhibit leaking behavior in any of the applications we studied.

V. CASE STUDIES

This section presents several case studies to demonstrate the resource leak problems we found in the studied applications. The leak problems were confirmed by code fixes written either by the application developers or by the authors. The detailed description of these defects can provide insights into possible new approaches for detection, diagnosis, and prevention of leaks in Android applications.

APV. As discussed earlier, there is a leak in native memory for APV. The application crashes with a large native memory footprint due to incorrect implementation in native memory reclamation. This crash is triggered by the neutral cycle $n_1 \rightarrow n_2 \rightarrow n_1$ in Figure 3. This defect cannot be easily discovered: examining the Java source code alone, the resource seems to be properly managed. It is also not easy to fix this problem due to lack of heap profiling tools for the native memory. In fact, after we discovered this defect during testing, we examined the code repository and observed that it took the developers several revisions to fix this problem. As native code and native memory are more heavily used in Android applications, compared to traditional server and desktop Java applications, new testing and diagnostic tools/techniques specifically targeting the usage of native memory are greatly needed for the Android platform.

ConnectBot. SSH client `ConnectBot` has a defect related to leaking of event listener objects. `TerminalView` represents the graphical interface of an SSH session, and it is a listener for font size changes. When `onStart` of `ConsoleActivity` is called, a new `TerminalView` is created and added to a container of listeners. However, it is never removed from the container. When `onStart` is called multiple times on the same `ConsoleActivity` object, `TerminalView` is leaked. One way to trigger this behavior is to start `ConsoleActivity` first, and then repeatedly go to the HOME screen and go back. Note that this leak cannot be triggered by rotation events, because a new `ConsoleActivity` is created whenever rotation occurs. This is an example showing why both ROTATE and HOME events should be considered for test coverage.

KeePassDroid. This password management tool saves user-provided login credentials in a password-protected database file, so that users can access them with one master password. Multiple database files can be maintained. When the application is first launched, it displays a list of database files in `FileSelectActivity` for the user to choose. When a database file is selected, a query is launched to retrieve the information in the file, and the result can be accessed through a `Cursor` object. The `Cursor` is remembered in a container so that it can be synchronized with the activity (i.e., it has the same lifecycle as the activity). The `Cursor` object is automatically cleaned up when its managing activity is destroyed. However, when we keep the same instance of `FileSelectActivity` alive, and come back to the selection list to select database files repeatedly, multiple `Cursor` objects would be saved in `FileSelectActivity`. Several crashing test cases are caused by this problem. A whole hierarchy of objects representing the query results are reachable from `Cursor` objects, leading to fast growth in memory consumption. A similar problem was also found in the `astrid` application. This is an important pattern to consider, because Android applications often interact with the built-in SQLite database, and `Cursor` objects are used to access the results of SQL queries. Testing the interactions between the application and the database is an important consideration for Android software development.

K9. In `K9`, a popular email client, a leak was discovered when rotating the screen after an email message is selected for display. Since it crashes after only a few repetitions of the ROTATE neutral cycle, this is an example of a leak that can be easily observed and thus cause negative user perception of the application. Heap snapshots suggest that a large number of objects are kept alive through a few `Thread` objects. The only code that creates threads is in `MessageView`, an activity that displays individual email messages. The threads are executed with the help of a thread pool executor, and because of this they are not explicitly started by calling `start` on them. In the standard library used by Android, a thread whose `start` method is never called is guaranteed to be leaking due to complex interplay between threads and thread groups. Since a leaking thread keep references to `MessageView`, a whole hierarchy of GUI objects are kept alive, leading to a quick crash. A simple fix is to create a `Runnable` object rather than a `Thread` object. Our understanding of the behavior of `Thread` and `ThreadGroup` was also confirmed by Android platform developers.[1] This is an example where a seemingly-innocent mistake (using `Thread` instead of `Runnable`), together with the unexpected behavior of the platform code, lead to problematic behavior. In fact, we have seen other leaks in the platform's management of resources (e.g., binders), in which case the application code does not have any defects, but still crashes. These observations highlight the need to repeatedly exercise resource-management code during testing, in order to expose unexpected interactions with the Android platform.

VLC. `VLC` is a popular cross-platform multimedia player. A leak was exposed when screen rotation is performed multiple times on its `AboutActivity`. This activity is implemented as a `FragmentActivity`, a new feature introduced in Android 4.0 and ported back to earlier versions. A fragment activity can manage `Fragments`, more flexible containers of GUI components. `Fragment` objects are created in `AboutActivity` and registered with the platform. Heap snapshots indicate that many fragments are kept alive. By default, the state of a `FragmentActivity` is (silently) saved and restored by the platform. In particular, all registered fragments are saved in memory before the activity is destroyed, and then restored from memory before it is recreated. Because of this behavior, `Fragment` objects created inside `AboutActivity` can never be garbage collected. For the developers this is an unexpected change to the management of an activity's lifetime, caused by this new Android feature. Subsequently we discovered that a later version of `VLC` disabled this default save/restore (by overriding relevant callback methods), which fixed the leak. This defect illustrates how new features introduced in the still-evolving Android platform can lead to defects due to poor understanding, which in turn motivates the need for regression testing and comprehensive strategies for test generation.

Application-specific neutral operations. We observed several examples where a leak is triggered by a neutral cycle with an application-specific functionality. For example, in the `astrid` task management application, database cursors are leaked

---

[1] `groups.google.com/d/topic/android-platform/y3G7v_U-hvA/discussion`

when the user changes the sort order of tasks, and then reverts back to the original order. Similarly, adding and then removing a task causes a cursor leak. The application-specific neutral cycles we consider for coverage are rather simple: they either involve a pair of neutralizing operations (e.g., add/remove, open/close) or a single operation that updates the displayed content (e.g., to refresh the current, unchanged list of email messages in K9). Currently, these cycles are provided as input to LEAKDROID. Automatic identification of such cycles, and perhaps even static analysis of their correctness with respect to leaks, are interesting problems for future work.

A. Discussion

Among the exposed defects, a diversity of resources are involved. Examples include not only traditional leaks such as memory leaks and thread leaks, but also Android-specific leaks such as binder leaks. Even for defects that exhibit the same "out-of-memory error" symptom, the underlying relevant resources could be different. Bitmaps, database cursors, and event listeners are some examples that fall into this category. Leaks caused by defects inside the Android platform and the standard library were also uncovered. This experience suggests that the proposed testing approach can effectively expose diverse resource leaks across a variety of applications.

Our experiments and case studies indicate that systematic model-based testing for resource leaks in Android software can be done effectively. They also point to interesting directions for future work. First, leak patterns based on neutral cycles can be leveraged to develop automated leak diagnosis tools. Based on our experience, we believe that a substantial part of the diagnosis process can be automated. It is particularly useful to identify strong correlations between the number of executed repetitions and the growth in the number of instances of certain classes. Allocation of and references to instances of classes that exhibit such correlations are usually related to the leak defects. Second, it is beneficial for understanding of resource usage/leaks to have analysis techniques that can automatically identify methods related to resource manipulation (e.g., allocation and reclamation). To achieve this goal, both static and dynamic analyses techniques may have to be developed. Finally, it is important to consider new mechanisms for prevention of resource leaks, with the help of better software abstractions and patterns for resource management.

VI. RELATED WORK

Memory leak detection and diagnosis. There exists a body of work on static analyses for memory leak detection. Typically these approaches target unmanaged languages such as C and C++. The few existing static leak detectors for Java can be very expensive, their false positive rates are not well understood, and it is unclear how to apply them to Android. There is also work (e.g., [34]) on static detection of resource leaks. These analyses typically require specification of resource management contracts/patterns; at present, it is an open question how they can be effectively used for Android software.

A practical alternative are dynamic analyses of memory leaks, both for managed (e.g., [7]–[11], [13]) and unmanaged languages (e.g., [12]). These approaches do not answer a key question: how can the leaking behavior be triggered during testing? An important contribution of our work is the insight that resource leaks in Android applications are often based on a few behavioral patterns, which allows targeted generation of tests to expose leaks of memory and other resources.

For memory leak debugging for Java, existing tools (e.g., [6]) can visualize and summarize object reference graphs from heap snapshots to help find unnecessary references. DePauw and Sevitsky [7] propose visualization for a period of time when temporary objects are expected to be created and released. LeakBot [8] formulates structural and temporal properties of reference graphs to detect memory leaks. Sleigh [9] encodes allocation/access sites in a single bit per object to enable low-overhead staleness detection. Cork [10] uses heap summarization to identify sustained growth of object instances. Leaks caused by unnecessary references stored in containers have been targeted through container profiling [11]. Xu et al. [13] target leaks due to repeated transactions, and use object lifetime assertions to identify leaking objects and their reference paths.

The proposed test generation strategy, based on coverage of various categories of neutral cycles, provides an effective mechanism for triggering the leak symptoms analyzed by all these diagnosis approaches. As discussed in Section III-C, the repetitive nature of the generated test cases presents interesting possibilities for custom variations of heap-differencing techniques (e.g., [8], [10]) and transaction-based leak diagnosis [13]. Future work may also consider generalizations of these approaches for analysis of native memory and resources other than memory.

Model-based GUI testing. Finite state machines and similar models for GUI testing have been used by a number of researchers (e.g., [15]–[25]). Given a GUI model, test cases can be generated based on various coverage criteria (e.g., [16]). In these approaches the focus is typically on functional correctness and the coverage criteria reflect this. In contrast, we are interested in non-functional properties, and the coverage categories we define explore specialized paths in the model (with multiple repetitions of a neutral cycle) in order to target common leak patterns. An alternative to model-based testing is random testing. For example, Hu and Neamtiu [27] use the Monkey tool [35] to randomly insert GUI events into a running Android application, and then analyze the execution log to detect faults. Random testing is highly unlikely to trigger the repeated behavior needed to observe sustained growth in resource usage.

Reverse engineering of GUI models has been studied by others (e.g., [17], [20], [29]) and has been applied to Android applications (e.g., [22]–[26], [30], [36]). Several techniques have been proposed to improve the precision of models and the test cases generated from them (e.g., [37]–[39]).

Testing and static checking for Android. Prior work has considered the use of concolic execution to generate sequences of events for testing of Android applications [40], [41]. Zhang and Elbaum [42] focus on testing of exception-handling code when applications are accessing unreliable resources. As an alternative to testing, static checking can identify various defects including invalid thread accesses [43], energy-related defects [44], and security vulnerabilities [45].

VII. Conclusions and Future Work

We propose a systematic and effective technique for testing of resource leaks in Android applications. Neutral cycles—sequences of GUI events that should not lead to increases in resource usage—are used to define test coverage criteria. Evaluation of this approach indicates that complicated and diverse resource leaks can be exposed by the generated test cases. These promising initial results suggest that such a testing technique is feasible and effective for detection of resource leak defects. Our investigation also points to several important directions for future work, including additional coverage criteria; better diagnosis techniques (e.g., by correlating repeated behavior with heap growth); increased focus on analysis of native memory as well as analysis of specific resources (e.g., database cursors, bitmaps); automated static or dynamic discovery/analysis of code that allocates and reclaims important resources; improved resource management through new software abstractions and patterns.

References

[1] comScore, Inc., "The great American smartphone migration," 2012, www.comscore.com/Press_Events/Press_Releases.

[2] "Stopping and restarting an activity," developer.android.com/training/basics/activity-lifecycle/stopping.html.

[3] P. Dubroy, "Memory management for Android applications," in *Google I/O Developers Conference*, 2011.

[4] "ConnectBot: Secure shell (SSH) client for the Android platform," code.google.com/p/connectbot.

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

[6] "Eclipse Memory Analyzer," www.eclipse.org/mat.

[7] W. DePauw and G. Sevitsky, "Visualizing reference patterns for solving memory leaks in Java," *Concurrency: Practice and Experience*, vol. 12, no. 14, pp. 1431–1454, 2000.

[8] N. Mitchell and G. Sevitsky, "Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications," in *ECOOP*, 2003, pp. 351–377.

[9] M. D. Bond and K. S. McKinley, "Bell: Bit-encoding online memory leak detection," in *ASPLOS*, 2006, pp. 61–72.

[10] M. Jump and K. S. McKinley, "Detecting memory leaks in managed languages with Cork," *Software: Practice and Experience*, no. 40, pp. 1–22, 2010.

[11] G. Xu and A. Rountev, "Precise memory leak detection for Java software using container profiling," in *ICSE*, 2008, pp. 151–160.

[12] J. Clause and A. Orso, "Leakpoint: Pinpointing the causes of memory leaks," in *ICSE*, 2010, pp. 515–524.

[13] G. Xu, M. D. Bond, F. Qin, and A. Rountev, "LeakChaser: Helping programmers narrow down causes of memory leaks," in *PLDI*, 2011, pp. 270–282.

[14] "Managing the activity lifecycle," developer.android.com/training/basics/activity-lifecycle.

[15] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences," in *ISSRE*, 2000, pp. 110–121.

[16] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for GUI testing," in *FSE*, 2001, pp. 256–267.

[17] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 884–896, 2005.

[18] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[19] Q. Xie and A. M. Memon, "Using a pilot study to derive a GUI model for automated testing," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 2, pp. 7:1–7:35, 2008.

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

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

[22] D. Amalfitano, A. Fasolino, and P. Tramontana, "A GUI crawling-based technique for Android mobile application testing," in *TESTBEDS*, 2011, pp. 252–261.

[23] 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*, 2012, pp. 258–261.

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

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

[26] "Android GUITAR," sourceforge.net/apps/mediawiki/guitar/index.php?title=Android_GUITAR.

[27] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," in *AST*, 2011, pp. 77–83.

[28] "Overview of sensors in Android," developer.android.com/guide/topics/sensors/sensors_overview.html.

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

[30] P. Tramontana, "Android GUI Ripper," wpage.unina.it/ptramont/GUIRipperWiki.htm.

[31] "Robotium testing framework for Android," code.google.com/p/robotium.

[32] "VuDroid project," code.google.com/p/vudroid.

[33] "Memory analysis for Android applications," android-developers.blogspot.com/2011/03/memory-analysis-for-android.html.

[34] E. Torlak and S. Chandra, "Effective interprocedural resource leak detection," in *ICSE*, 2010, pp. 535–544.

[35] "Monkey: UI/Application exerciser for Android," developer.android.com/tools/help/monkey.html.

[36] 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*, 2012, pp. 93–104.

[37] X. Yuan and A. M. Memon, "Generating event sequence-based test cases using GUI run-time state feedback," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 81–95, 2010.

[38] X. Yuan, M. B. Cohen, and A. M. Memon, "GUI interaction testing: Incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559–574, 2011.

[39] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, and A. M. Memon, "Lightweight static analysis for GUI testing," in *ISSRE*, 2012, pp. 301–310.

[40] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *FSE*, 2012, pp. 1–11.

[41] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *ISSTA*, 2013, pp. 67–77.

[42] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," in *ICSE*, 2012, pp. 595–605.

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

[44] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?" in *MobiSys*, 2012, pp. 267–280.

[45] 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.