Characterizing and Detecting Resource Leaks in Android Applications

Chaorong Guo*‡, Jian Zhang*, Jun Yan†, Zhiqiang Zhang*‡, Yanli Zhang*‡
*State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences, Beijing, China
{guocr, zj, zhangzq, zhangyl}@ios.ac.cn
†Technology Center of Software Engineering
Institute of Software, Chinese Academy of Sciences, Beijing, China
yanjun@otcaix.iscas.ac.cn
‡ University of Chinese Academy of Sciences, Beijing, China

***Abstract*—Android phones come with a host of hardware components embedded in them, such as Camera, Media Player and Sensor. Most of these components are exclusive resources or resources consuming more memory/energy than general. And they should be explicitly released by developers. Missing release operations of these resources might cause serious problems such as performance degradation or system crash. These kinds of defects are called *resource leaks*. This paper focuses on resource leak problems in Android apps, and presents our lightweight static analysis tool called *Relda*, which can automatically analyze an application's resource operations and locate the resource leaks. We propose an automatic method for detecting resource leaks based on a modified Function Call Graph, which handles the features of event-driven mobile programming by analyzing the callbacks defined in Android framework. Our experimental data shows that Relda is effective in detecting resource leaks in real Android apps.**

***Index Terms*—Android apps; resource leak; static analysis;**

I. Introduction

To enrich the user experience, Android phones come with a host of hardware I/O components embedded in them, such as Media Player, Camera and Sensor, most of which are exclusive resources or resources consuming more memory or energy than general. These resources require explicit management. Absence of their release operations results in *resource leaks*, which may lead to performance degradation (e.g., huge energy or memory consumption), or even system crash.

Unfortunately, resource management bugs are common in Android programs, for which a number of reasons should be responsible. Firstly, unlike traditional programming languages, the Android framework transfers the burden of resource management to the developers, but the developers might omit a call to release a resource, on the assumption that Android framework would release the resource automatically. Secondly, developers tend to focus on the functionalities, the performance-related problems will not be their concerns. And they often don't conduct performance testing before they release the applications. Thirdly, programmers need to understand all relevant API contracts, on which they could easily make incorrect assumptions. Finally, even a scrupulous programmer can easily fail to release all resources along all possible invocation sequences of event handlers.

In this paper, we focus on resource leak problems in Android applications. We have built a lightweight static analysis tool Relda, which can automatically analyze an application's resource operations and detect the missing release operations to help programmers identify their root causes. Klocwork [21] and FindBugs [14] are static code analysis tools, which provide analysis of Java source code or Java bytecode respectively. Klocwork is a commercial software that delivers comprehensive source code analysis solution and complete codebase inspection for C++, C, C# and Java. It hasn't provided analysis of bytecode so far. It inspects a few Android-related resources [22], which are not properly released after use, such as Media player and Camera. FindBugs is a defect detection tool for Java that uses static analysis to look for bug patterns, such as null pointer dereferences, infinite recursive loops, bad uses of the Java libraries and deadlocks. It also adds detectors for Android specific coding, but the detectors FindBugs provides only inspect unclosed Cursor instance and unclosed stream of Android [15]. Compared with these works, our work provides a strategy that focuses on Android-platform-related resources, with Android platform and language features considered. Our target-specific tool is more comprehensive and precise.

Our analysis focuses on three kinds of resources: exclusive resources, memory-consuming resources and energy-consuming resources. Their common characteristic is that they should be released explicitly as specified in the Android API Reference [1]. We propose an automatic solution to detecting resource leaks based on a modified Function Call Graph (FCG) analysis, which handles the features of event-driven mobile programming by analyzing the callbacks defined in Android framework. We first collect a near-complete resource table after examining through the relevant 313 classes extracted from all the 1944 classes in the Android API Reference. The table includes all the resources that the reference requires manual release. And then we describe how we automatically

[1]http://developer.android.com/reference/packages.html

978-1-4799-0215-6/13/$31.00 © 2013 IEEE

detect resource leaks in Android apps by analyzing their bytecode. Finally, we provide experimental data showing the effectiveness of our tool in detecting various resource leaks in real apps.

The rest of this paper is organized as follows: In Section II, we present our resource leak analysis and detection technique and illustrate how it works. Experimental results are shown in Section III. Section IV discusses the related works. In Section V, we give the conclusions and discuss the future work.

II. Methodology

Before making further discussions, we first give the definitions of three terms used in this paper:

- **isolated/entry points**: Functions that are not invoked by any other function in the program.
- **callbacks/handlers**: Methods invoked when a user event is triggered or a certain system state arrives, which is a feature of event-based programs.
- **children**: Functions which are invoked by a certain function.

In preparation for our study, we collected some leak problems in Android apps from mobile app forums and discussion groups, and we found that many users complained about problems of memory shortage, energy drain and application crashes, most of which are relevant to the use of Camera [8] [9] or Sensor components [29] [30] [31]. Then we checked the Android API Reference about these components, and categorized the resources into three classes:

- If a resource is an exclusive resource, a missing release operation will cause other applications' endless waiting for it. For the Camera component, the manual says [7]: *"Call release() to release the camera for use by other applications. Applications should release the camera immediately in onPause() (and re-open() it in onResume())."*
- As for the Media Player component, the reference says [24]: *"It is recommended that once a MediaPlayer object is no longer being used, call release() immediately so that resources used by the internal player engine associated with the MediaPlayer object can be released immediately."* This is a memory-consuming resource, which consumes more memory than general resources.
- Another kind of resources are energy-related resources. For example, SensorManager lets you access the device's sensors through registering it to the SENSOR_SERVICE. We should cancel the registration for the SensorManager object when we don't need it. As the manual says about the Sensor [32]: *"Always make sure to disable sensors you don't need, especially when your activity is paused. Failing to do so can drain the battery in just a few hours. Note that the system will not disable sensors automatically when the screen turns off."*

In this paper, we aim to detect resource leaks, i.e., situations where resources should be released manually by the developer but he/she failed to do so. Figure 1 provides a high-level overview of our approach. It consists of several key parts. We first disassemble an application to get the dalvik bytecode [11] using reverse engineering technique. Then we traverse the bytecode in sequential order to construct the FCG, which contains analysis of resource operations and callbacks defined in the Android framework. Finally we use depth-first search on the FCG and analyze the class hierarchy to identify unreleased resources.

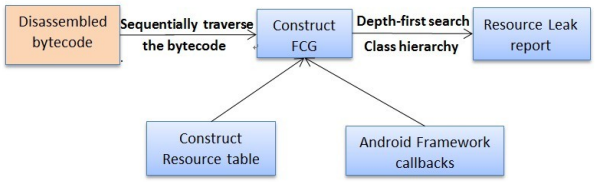

Fig. 1. High-level Overview of Our System

A. Key Resource Operations

1) Extract Relevant Classes: We downloaded the Android developer reference manual, and searched the summary section of methods of all the 1944 classes in reference folder with the modification of the keywords (Table I), e.g., the keyword "unregister" is modified to the style "unregister*(", which means matching any member method whose name starts with the word "unregister". For example, "unregisterListener(" in SensorManager is one such case.

TABLE I
KEYWORDS TO MATCH RESOURCE RELEASE METHODS OF CLASSES

abandon	cancel	clear	close	end
disable	finish	recycle	release	remove
stop	unload	unlock	unmount	unregister

2) Check Relevant Method Descriptions: For all the 313 classes which contain a member method whose name starts with one of these keywords, we manually check the detailed description of all these methods to figure out if the requested resource should be released manually. After that, we read through the training section of relevant classes manually, and figure out the appropriate event routine (e.g., `onPause()`) of activities to release these resources. And we finally got 65 resource operations which involve resources that need to be released manually. These frequently used are summarized in Table II.

B. Event Handlers

The features of event-driven mobile programming make our task complicated. In particular, due to the large number of callbacks used by the Android framework, Android apps essentially expose a set of callbacks to the system instead of a single "main method". Our approach leverages the knowledge of how these callbacks are defined in Android to identify them. We divide them into two categories:

TABLE II
SUMMARY OF FREQUENTLY-USED RESOURCE REQUEST AND RELEASE OPERATIONS

Resource package	Resource name	Resource operations	Suggested place to release
android/media	AudioManager	requestAudioFocus/abandonAudioFocus;	`onPause()`
	AudioRecord	new/release;	`onPause()`/`onStop()`
	MediaPlayer	new/release; create/release; start/stop;	`onPause()`/`onStop()`
android/hardware	Camera	lock/unlock; open/release; startFaceDetection/stopFaceDetection; startPreview/stopPreview;	`onPause()`
	SensorManager	registerListener/unregisterListener;	`onPause()`
android/location	LocationManager	requestLocationUpdates/removeUpdates;	`onPause()`
android/os	PowerManager.WakeLock	acquire/release;	`onPause()`
	Vibrator	vibrate/cancel;	`onDestroy()`
android/net/wifi	WifiManager.WifiLock	acquire/release;	`onPause()`
	WifiManager	enableNetwork/disableNetwork;	`onDestroy()`

1) Callbacks triggered automatically when a certain system state occurs according to Android framework's definition. We call them system handlers (system callbacks) for short.
2) Callbacks triggered by user events. We call them user handlers (user callbacks).

1) System Handlers: Instead of a traditional "main method" of some kind, Android apps contain one or more components defined in its manifest file. Each component can potentially define multiple entry points. Figure 2 shows the important state paths of an Activity [1]. The colourless round rectangles represent callback methods you can implement to perform operations when the Activity moves between states. The entire lifetime of an activity happens between the first call to `onCreate(Bundle)` through to a single final call to `onDestroy()`. The visible lifetime of an activity happens between a call to `onStart()` until a corresponding call to `onStop()`. The foreground lifetime of an activity happens between a call to `onResume()` until a corresponding call to `onPause()`. The `onCreate()`, `onStart()` and `onResume()` callbacks are defined entry points of an activity, the exit points are `onPause()`, `onStop()` and `onDestroy()`.

Apart from them, there are a large number of callbacks defined in the Android framework. App executions often pass through the framework to emerge elsewhere in the app. For a concrete example, consider the `java.lang.Thread` class. The `run()` method is a common callback defined in `java.lang.Thread`. A developer can simply extend this class, implement the `run()` method, and then call the `start()` method to schedule the thread. But when we analyze the code within the app, the `run()` method does not appear to be reachable from the `start()`, despite the fact that after the `start()` method is called, control flow goes eventually to the `run()` method during the runtime (Figure 3). Permission examining tool Woodpecker [41] leverages these well-defined semantics in the Android framework APIs to link these two methods directly in the control flow graph (CFG), resolving the discontinuity in the CFG construction.

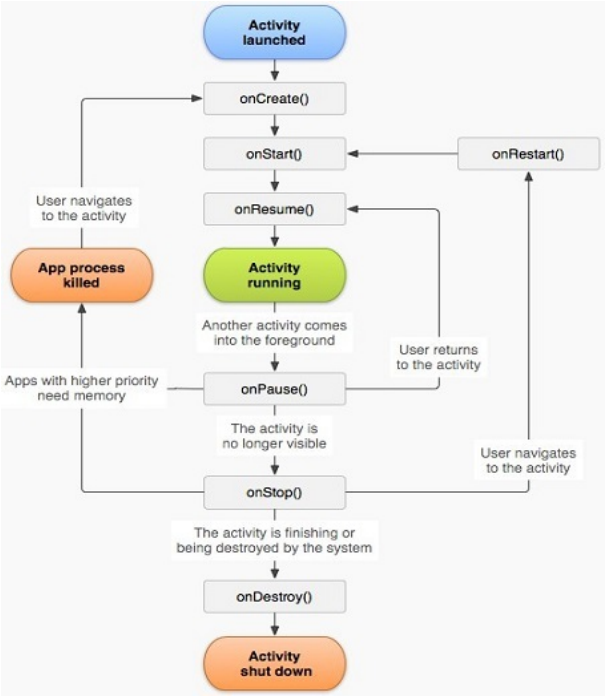

Fig. 2. Activity Lifecycle Diagram

And it applies this strategy to a number of callbacks (e.g., message queues, GPS). According to our studies there are lots of other callbacks defined in the Android framework. For example, the camera service will initiate a series of callbacks to the application as the image capture progresses.[2] The `shutter` callback occurs after the image is captured. This can be used to trigger a sound to let the user know that the image has been captured. The `raw` callback occurs when the raw image data is available. The `postview` callback occurs when a scaled, fully processed postview image is available. The `jpeg` callback occurs when the compressed image is available. And a programmer can also define callbacks by

[2]They are passed as arguments within the takePicture method of the Camera class.

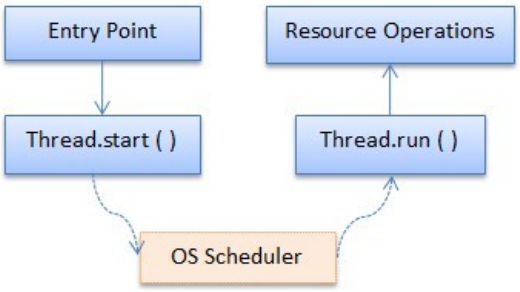

Fig. 3. A Discontinuity in the Control Flow Introduced by the Android Framework

implementing user-defined interfaces. These callbacks will be invoked automatically without user interactions, so we classify them as system handlers too.

In our approach, we maintain a list to record the registration of all callbacks (except for user callbacks, as described below), then connect it to the callback itself when constructing FCG.

2) User Handlers: The other kind of events is usually an UI event. Android provides two mechanisms handling them: Mechanism based on registered callbacks and based on event listeners (which are also a kind of callbacks). These events contain click and touch events (e.g., onClick, onLongClick, onTouch, onTouchevent), keyboard events (e.g., onkey), state change events (e.g., onFocusChange, onItemSelected) and so on. Their common characteristic is that whether it will be called or not is determined by the user's action. For example, in function `onCreate()`, we register two listeners respectively for button A and button B, and then request a resource in the `onClick()` method which is triggered by a click on button A, and the corresponding release operation is carried out in another `onClick()`, which is controlled by button B. This is a potential resource leak, because if the user just clicks button A, and the app is interrupted by another activity started by the user or system (like switching to the home screen, or an incoming call), then the resource requested before is not released. So in our scheme, we don't connect the registration of the user callback to the callback itself, which indicates a normal case (the user clicks on Button A to request a resource, then clicks on Button B to release the resource). So these user event functions are still entry points in our FCG.

C. FCG Construction

Within an application, we inspect all the resource operations to collect the potential resource leaks it contains. For a local resource (resource declared in a function), we suppose the developer will release it when the method finishes. For a global resource (resource declared in a class outside all the functions), it should be released on all the paths that can be reached from its request point to the end of path, or in the exit point function (for activity or service class or its inner class), or in at least one callback of all interfaces implemented by the classes, as described in Figure 4. *UT* represents user handlers, *ST* represents system handlers and *N* represents normal functions.

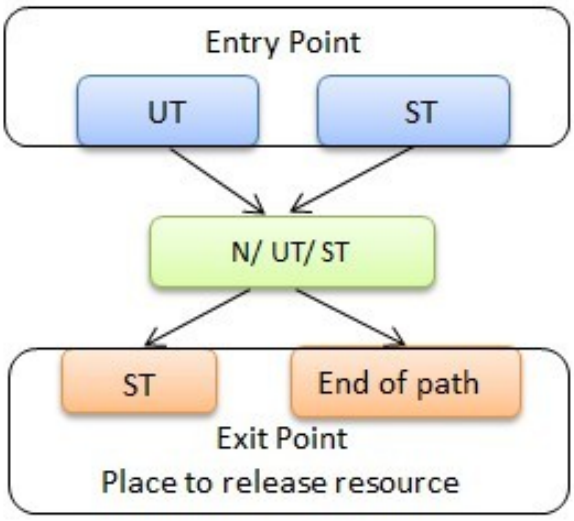

Fig. 4. Feasible Invocation Orders of Event Handlers

The round rectangle in green represents a place to request a resource, and the rectangle in orange represents a place to release a resource.

We use a free reverse engineering tool *androguard* [3] to generate standard dalvik bytecode of an app. Then we traverse the bytecode in sequential order. In the bytecode, it is very easy to know the function calling relationships within an app according to the "*invoke*" instructions, especially for methods with parameters. The definition and the invocation of a method have the same parameter declaration, while, in the source code, the formal parameter in method definition is not the same as the actual parameter in method invocation. And for each resource request and release operation, we check instructions starting with "*iget-object*" and "*iput-object*" to find the resource object's reference, then we use the method described in Section II-B to handle event-driven calls in Android applications, which adds links for system callbacks. Finally, we obtain an entire FCG.

Figure 5 is a code snippet of an activity, and figure 6 shows an example FCG of the activity. In figure 6, the round rectangles in orange represent functions of the activity. The solid arrows represent calling relationship between calling functions and called functions. `Camera.open()` and `mCamera.release()` are resource request and resource release operation respectively. `onCreate()`, `onPause()`, `onShutter()` and `onPictureTaken()` are system callbacks, `onClick()` is a user callback and `setOnClickListener()`, `takePicture()` and `mCamera.takePicture()` are normal functions, as we described in Figure 4.

D. Resource Leak Summary and Report

Through the above process, we build the FCG of an app. And we obtain the `entry_points_list` (whose items are functions which are not invoked by any other function), each function's `op_list` (whose items are either

```
1.public class myCamera extends Activity{
2.  public void onCreate(Bundle savedInstanceState){
3.    mButton.setOnClickListener(
                  new Button.OnClickListener()
4.    {  public void onClick(View v){
5.         mCamera = Camera.open();
6.           takePicture();
7.       }
8.    });
9.  }
10.  private void takePicture(){
11.    mCamera.takePicture(shutterCallback,
12.                  rawCallback,jpegCallback);
13.  }
14.  private ShutterCallback shutterCallback=new
15.    ShutterCallback(){
16.    public void onShutter(){
17.    }
18.  };
19.  private PictureCallback rawCallback=new
20.    PictureCallback(){
21.    public void onPictureTaken(){
22.    }
23.  };
24.  private PictureCallback jpegCallback=new
25.    PictureCallback(){
26.    public void onPictureTaken(){
27.    }
28.  };
29.  protected void onPause(){
30.    mCamera.release()
31.  }
32.}
```

Fig. 5. Code Snippet of a Sample App

resource operations or child function calls) and each function's `children_list` (whose items are functions which are invoked by this function). The resource summary algorithm (summarized by Algorithm 1) takes them as the input. It uses depth-first search to analyze the resource operations and child function calls in all the entry points. During the searching process, we use a few extra structures to mark functions that have been analyzed before to avoid endless loop when dealing with recursions and loops. We traverse each recursive function or loop just once. The process will eventually generate resource summaries of all entry points. And we also provide a series of UI commands that can be used to query extra information. You can input any function you are concerned with (e.g., `onCreate()`) to check the resource operations in this function and you can also query the calling sequences between two functions by simply providing the function's label.

Algorithm 1 Resource summary of all entry points

Input: $entry\_points\_list$, all functions' op_list and $children\_list$
Output: all entry points' res_list

```
for each entry_point ∈ entry_points_list do
  if entry_point.children_list = ∅ then
    entry_point.res_list = entry_point.op_list
  else
    stack = reverse(entry_point.op_list)
    while stack ≠ ∅ do
      op_item = stack.pop()
      if op_item ∈ entry_point.children_list then
        child_node = get_obj(op_item)
        if child_node.op_list ≠ ∅ then
          stack.push_all(reverse(child_node.op_list))
      else
        if is_res_request(op_item) then
          entry_point.res_list.append(op_item)
        else if is_res_release(op_item) then
          if request(op_item) ∈ entry_point.res_list
            then
              entry_point.res_list.delete(
                    request(op_item))
          else
            entry_point.res_list.append(op_item)
```

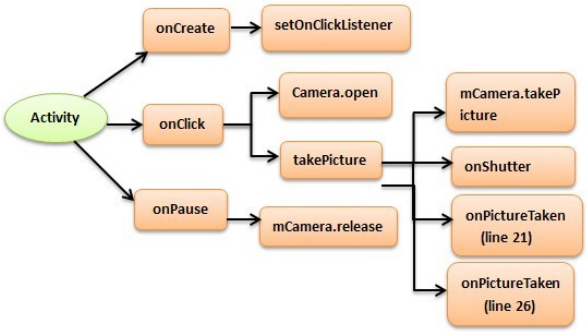

Fig. 6. An Example FCG of an Activity

By observing the resource summaries of entry points, we find lots of system and user handlers exiting in inner classes of an activity or service. Inner classes are characterized by '$' in dalvik bytecode, which can be easily identified. As for resource operations in an activity or service or its inner class, if they are not released as soon as they are not needed, we extra check if a resource request is released in exit point functions (e.g., `onPause()`) of its corresponding activities or services. For resource requested in an ordinary class, we scan all interfaces it implements. Then we check if the resource is released in at least one callback of each of them. As described in Figure 7, the programmer requests audio focus in class `AssistantTtsPlayer` and abandons the focus in system callbacks (`onComplete()` and `onError()`) of all interfaces implemented by `AssistantTtsPlayer`. It is the same situation when dealing with Camera, which usually implements the `surfaceHolder.Callback` interface. We can release the camera resource in its `surfaceDestroyed` callback.

III. Evaluation

We now present experimental results of resource leak detection using our tool Relda. We first present a summary of

TABLE III
Resource Leaks in 55 Demo Apps

App	Unreleased Resource	Request Method
myWifi	WifiManager	enableNetwork
myCamera	Camera	open
myAudioManager	MediaPlayer	create
mySeekPlayer	MediaPlayer	create

TABLE IV
SUMMARY OF EXPERIMENTED APPS WITH RESOURCE LEAK

Apps	Bug Description	Ref
Agenda Widget 1.6.17	PowerManager.Wakelock is not released in several classes in package `widget`	[2]
k9mail 4.006	PowerManager.Wakelock is not released in `MessagingControllerPushReceiver`	[20]
Osmdroid r751	`OpenStreetMapActivity` should enable/disable location updates in `onResume()`/`onPause()`	[25]
Baidu Map 1.2 *	The listener for SensorManager is not released in package `ar` It is also not released in all cases of received messages in `onCreate()`	[5]
Baidu Voice Assistant 2.1 *	Audio focus is not released in `MusicService`	[6]
Baidu IME 3.5.1.4 *	Camera requested in `CaptureActivity` is not released	[4]
Google Email 2.3.4.1 *	In several activities in package `gm`, ContentProviderClient is not released	[18]

```
1.public class AssistantTtsPlayer implements
2.    TTSPlayerCompleteListener, TTSPlayerErrorListener
3.{
4.  mAudioManager = ((AudioManager)mContext.
          getSystemService("audio"));
5.  mAudioManager.requestAudioFocus();
6.  public void onComplete()
7.  {
8.    mAudioManager.abandonAudioFocus();
9.  }
10.  public void onError(int paramInt)
11.  {
12.    mAudioManager.abandonAudioFocus()
13.  }
14.}
```

Fig. 7. Release Resource in One Callback of Each Interface

the detection results on 98 Android apps, and then discuss the causes for the reported leaks and overhead of the scheme.

A. Experiment Setup

We conduct our experiments in the following steps:

- First, we collected the source code of 55 apps from an Android application development book [36] and 80 real app installers (*.apk* files) from Google Play[3] or from individual official app sites.
- For the .apk installers, we decompile the apps from the installers to Java source code using Dare [12] [39], and successfully got 43 decompiled source code[4] (decompilation failed for the remaining apps). Thus we got $55+43 = 98$ apps with source code or decompiled source code.

B. Resource Leak Reports

Our experimental subjects are the 98 .apk installers. Relda takes them as input. After it finishes running, each app's resource leak items are reported, as well as log files that record where each resource is requested and released. For 55 apps in the development book, all the 4 leaks reported by our tool have been confirmed manually by checking their source code directly. These resource leaks are about different kinds of resources, but the patterns are similar. The developer tries to release resources in user handlers which are not necessarily invoked before the app exits. The resource leak report is shown in Table III.

[3]https://play.google.com/store
All of them are top free apps, and most are used daily.

[4]Not all app installers could be transformed into (meaningful) source code, especially the ones that have been obfuscated during compilation (using tools like proguard [27]).

For the 43 apps that we successfully get decompiled source code, 25 of them have one or more resource leaks, and there are 92 reported leaks in total. Table IV presents typical resource leaks we found in several popular apps, including confirmed instances of resource leaks in Osmdroid [26], Agenda Widget and k9mail [37], as well as some instances that are not confirmed (Entries with *), e.g., resource leaks in Google Email [16] [17] and Baidu Maps [29] [30] [31].[5] The whole resource leaks summary are listed in Table V. We use a standalone graphical utility JD-GUI [19] to display Java source codes of these ".class" files obtained by Dare. For these relatively large apps, it is not possible to manually enumerate all possible paths to verify the correctness of the tool due to exponential explosion. In our evaluation, for each of the resource leaks reported by Relda, we check the decompiled Java source code manually if it's a real leak. By analyzing some relatively small apps and possible shortages of our methodology, we infer certain patterns (e.g., if-else, cases of received messages) that may incur false negative (a resource leak Relda doesn't report but it really is). So we check these points of decompiled Java source code additionally.

TABLE V
SUMMARY OF 10 MOST COMMON RESOURCES UNRELEASED IN APPS

Unreleased Resource	Request Method	App	Count
MediaPlayer	new/create	11	24
MediaPlayer	start	8	15
VelocityTracker	obtain	7	11
PowerManager.WakeLock	acquire	6	12
WifiManager	enableNetwork	2	8
Vibrator	vibrate	5	10
ContentResolver	acquireContentProviderClient	2	2
AndroidHttpClient	newInstance	2	2
SensorManager	registerListener	2	4
AudioManager	requestAudioFocus	1	4

[1] The third column represents the number of apps containing unreleased resources.
[2] The fourth column represents total number of the unreleased resources.

We classify the leaks into three categories: (a) True Positive (TP): Resource leaks reported by the tool are also found manually by checking the source code; (b) False Positive (FP): Leaks reported by the tool but we do not find manually; (c) False Negative (FN): Leaks we found manually but the tool did not report, as we mentioned above. Table VI summarizes the results. Explanations of each entry will be discussed in Section III-C. We found 81 leaks in the TP set, 11 in the

[5]We have reported the relevant bugs to Google and Baidu, but have not yet received any feedback.

FP set and 8 in FN set. So the observed analysis precision is 88.0%, and the recall is about 91.0%.

TABLE VI
SUMMARY OF LEAKS BREAKDOWN

Causes of Leaks	**Total** 92
Released resources in user handlers	27
Forgot to release a resource	19
Didn't release in callbacks of all interfaces	14
Released in improper event routine (e.g.,onDestroy)	13
False negatives	8
False positives	11

C. Resource Leak Classification by Causes

1) Released Resources in User Handlers Which May Not Be Invoked (27): The largest category of leaks are of this kind. In this case, the developer tries to release resources in user handlers which are not necessarily invoked before the app exits. In addition to the 4 leaks found in the development book, this kind of leaks also appears in real apps we downloaded from Android market. For example, in Baidu Map 5.0.0, the programmer tries to release LocationManager and AudioManager resources only in user handlers (e.g., onKey, onClick). A good suggestion is to release them in their corresponding exit point function, as we point out in Table II.

2) Forgot to Release a Resource (19): The cause of these leaks is that the programmer simply forgot to release a requested resource throughout the code. Although it seems like a simple mistake, this does happen in real apps. For example, in Google Email 2.3.4, ContentProviderClient [10] is not released in several activities in package `gm`. But all the resource leaks about ContentProviderClient have been repaired in Google Email 4.3.1. In Baidu Map 1.2 and Baidu Map 4.3, the programmer forgot to release the listener for SensorManager in package `ar` (Augmented Reality). The resource leak has been repaired in Baidu Map 5.0.0.

3) Failed to Release a Resource in Callbacks of All the Implemented Interfaces (14): There are many different executable paths invoked by different user actions and hardware states which depend on the external environment. Even a careful programmer can easily fail to release all resources along all possible invocations of event handlers. For resource requested in an ordinary class, we extra check if it's released in at least one callback of all interfaces implemented by the classes. As we described in Section II-D. In youku 3.0 [35], the programmer only tries to close PowerManager.WakeLock resource object in callback `onFinished()` of class `DownloadListenerImpl`. This is not enough. The suggested modification is to close the resource in `onCancel()` and `onException()` too.

4) Didn't Properly Understand the Lifecycle of Android Apps (13): In this case, programmers release resources only when the app finally exits, in `onDestroy()` of the activity. In Android, an app activity once started is always alive. When the user exits any app, Android saves the state of the app and passes it back to the app if the user returns to it. The app is only completely killed when the phone is critically low on RAM or when the app kills itself. This methodology is used to reduce the startup time of the app and to maintain its state. This essentially means that the app may not actually be destroyed for a very long period of time. `OnDestroy()` is only called when the app component is about to be destroyed, but many app developers try to release the requested resources in the `onDestroy()` callback, instead of in `onPause()`. As a result, once an app with this leak is started, the phone will finally exit when it is running critically low on memory (which may take a long period of time). For a concrete example, in open source project Osmdroid [25], our tool reported a resource leak resulted from the delayed unregistration of the location listener [26] [47]. Figure 8 gives a simplified version of the concerned code. When `MapActivity` is launched, it gets a reference `mManager` to system location manager and registers a location listener `mListener` with the Android system (Line 13), `mListener` is registered to listen to users location changes (Lines 7-11). If the Android system plans to destroy `MapActivity` (Lines 15-18), `mListener` would be unregistered (Lines 17). If Osmdroid's users switch their Android phones to another application, `MapActivity` would be put to the background (not destroyed), and the registered `mListener` would still keep running for location sensing. All location data would be used to refresh an invisible map. Then, a huge amount of energy and memory would be wasted.

```
1.public class MapActivity extends Activity{
2.  protected LocationListener mListener;
3.  protected LocationManager mManager;
4.  public void onCreate(){
5.    //get a reference to system location manager
6.    mManager = getSystemService(LOCATION_SERVICE);
7.    mListener = new LocationListener() {
8.      public void onLocationChanged(Location loc) {
9.        ...
10.     }
11.   }
12.   //GPS listener registration
13.   mManager.requestLocationUpdates(GPS,...,mListener);
14. }
15. public void onDestroy(){
16.   //GPS listener unregistration
17.   mManager.removeUpdates(mListener);
18. }
19.}
```

Fig. 8. Resource Leak in Osmdroid Application (Issue 53)

5) False Negatives (8): Without considering intra-procedural flow analysis, certain patterns of program statements incur false negatives, e.g., if-else, cases of received messages. According to our case studies, false negatives regularly occur when handling received message problems. In this case, a resource should be released in all cases of received messages. But our tool will not report a leak as long as it's released in one case of them. To solve this problem, we modify our tool to present directly information about the resource operations in all cases and whether a resource is released in all cases. We find several real apps having this kind of resource leaks. For example, Baidu Map 1.2, the programmer tries to release a registerListener for SensorManager in one case of the received message in

```
1. public static void onCreate{
2.   demoStart();
3.   demoHandler = new Handler(){
4.     public void handleMessage(Message message){
5.       demoStop();
6.     }
7.     // call demoStart() here;
8.   }
9. }
```

Fig. 9. False Positive Due to Registration Orders of Callbacks

handler `AndroidJni.w`. The leak is repaired in Baidu Map 4.3. We will consider using Soot [33] or WALA [34] to solve the problem of intra-procedural flow analysis in our future work.

6) False Positives (11): 11 of 92 leaks are reported to contain a leak, but upon further manual analysis, they turn out to be false positives. There are two major reasons for the false positives in the 11 reported leaks.

One reason is useless isolated points that are not invoked by any other functions. These functions are treated as entry points in our previous analysis, but upon further manual analysis, we find they are just useless functions. They may come from programmers' carelessness for not wiping out the useless code.

Another reason for false positives is the registration orders of callbacks. As an example in Figure 9, programmers usually write the resource operations in the following order: Request a resource in method `demoStart()`, then release it in `demoStop()`. When using our method described in Section II-B1, i.e., connect the registration of a system callback to the callback itself, the requested resource in `demoStart()` will be analyzed to be released in `demoStop()` during the depth-first search. But if a programmer places the `demoStart()` (Line 2) to the new place (Line 7), the modification will incur a reported resource leak. From this example, it seems that we can solve the problem by simply postponing the analysis of the registered handler, e.g., put it to the end of `onCreate()`. We do not repair this for two reasons: It may occur that a resource request rather than a resource leak in `Handler()` is invoked; There may be a lot of other callbacks, that also need to postpone the analysis. In fact, with detailed resource operations information provided, we easily locate the suspicious statement and figure out the false positives with a bit of manual work.

D. Overhead of the Scheme

Our tool conducts light-weight static analysis without semantic parsing of the program, and focuses only on our interested resource operations, thus the analysis overhead is considerably low. Table VII presents our tool's analysis overhead for seven popular apps. The table shows the number of classes and methods of each app and the time spent to run the resource leak analysis. (The time does not include the decompilation time of the .apk files.[6]) Our analysis finishes within about 1 minute for each of the 7 apps.

TABLE VII
OVERHEAD OF RESOURCE LEAK ANALYSIS FOR 7 POPULAR APPS

Apps	Classes	Methods	Analysis time (seconds)
MyTrack 2.0.3	1037	5882	15
Agenda Widget 1.6.17	629	2624	7
k9mail 4.006	1025	7733	35
Baidu Map 1.2	345	1447	6
Baidu Voice Assistant 2.1	1195	8199	58
Baidu IME 3.5.1.4	420	3804	61
Google Email 2.3.4.1	1512	10800	43

IV. RELATED WORK

A. Resource Leak Detection in Java Programs

Resource leak detection is a classic problem in program analysis. It is hard for developers to avoid all resource leaks. There are two kinds of methods, dynamic and static analysis, that provide inspection for them. QVM [42] suggests an approach to detect defects by using a specialized runtime environment, and it enables the checking of violations of user-specified correctness properties. Static analysis tools like FindBugs [14] and Klocwork [21] detect a wide range of problems including a few Android-platform-related resource leaks [15] [22]. Tracker [40] performs inter-procedural static analysis to find resource leaks in Java programs, and ensures that no resource safety policy is violated on any execution path. Since Android applications update quickly, we adopt the light-weight static analysis method, that can scan an industrial-level Android package efficiently. Comparing to those works, our work focuses on Android-platform-related programming features and resources, and our target-specific tool is more comprehensive.

Among the studies of Smartphone resources, energy related issues have received attentions for the past several years. Some of them estimate energy consumption of an application or its components. Pathak et al. proposed EProf [38], which is a fine-grained energy profiler for smartphone apps. It breakdowns the total energy consumed into per-component energy consumption by tracking the activities of energy-consuming entities. And it also gives insights into energy breakdown per thread and per routine of the app. There are also some works that analyze energy-related defects in Smartphone applications. Yepang Liu et al. built a tool called GreenDroid [47] on top of Java PathFinder to find energy inefficiency problems. Similar to those works, our work focuses on detecting resource leak problems including several energy-related resources, so it can also be used to detect energy inefficiency problems caused by incorrect release (or absence of release) of energy-consuming resources.

[6]Androguard supports three decompilers (DAD, dex2jar + jad, ded), we used its default decompiler DAD, which is very fast due to the fact that it is a native decompiler, and the decompilation is done on the fly for each class/method.

B. Memory Usage Problem

Memory usage problem is a long studied topic. In traditional programming languages (e.g., C), memory recycling task is done by the program itself, the program allocates and frees the memory occupied by variables explicitly. There are some works on detecting memory leaks, e.g., we proposed a static analysis tool Melton in our previous work [49]. While the main part of Android application program is realized using Java programming language, an outstanding feature of Java language is that most memory transactions are managed through Java virtual machine (JVM) and garbage collection (GC) mechanism. Java programmers allocate memory simply by creating objects, and don't need to care about how and when the objects should be recycled. But leak may happen when you maintain reference to an object that prevents GC. So there are inevitable memory leak problems in Java program.

MAT [23] is an Eclipse Memory Analyzer, which is a fast and feature-rich Java heap analyzer that analyzes productive heap dumps with hundreds of millions of objects. It quickly calculates the retained sizes of objects, sees who is preventing the garbage collector from collecting objects, and generates a report to automatically extract leak suspects. In paper [48], the authors analyzed reasons of Android application memory leaks and introduced how to avoid memory leaks when compiling application program using MAT. Other commercial leak detectors such as [13] [28] enable visualization of heap objects of different types. Existing academic tools use growing types [45] [46] (i.e., types whose number of instances continues to grow) or object staleness [43] [44] to identify suspicious data structures that may contribute to a memory leak. But all of them gather memory-related information using dynamic analysis techniques. They don't have an explicit definition of resources, and don't have the ability to pinpoint the cause of a memory leak.

V. Conclusion and Future Work

This paper makes a comprehensive analysis of resource leak problems in Android apps, and describes how we automatically detect resource leaks in Android apps. We first obtain a near-complete resource table which includes almost all the resources that the reference manual requires to release manually. The table can also be expanded to define other concerned resources. Then we propose an automatic solution to detecting resource leaks based on a modified Function Call Graph (FCG), and it handles the features of event-driven mobile programming by analyzing the callbacks defined in Android framework. We build a light-weight static analysis tool Relda, it focuses on Android-platform-related programming features and resources, and incurs reasonable overhead and false positives. Further more, with detailed resource operations information provided by our tool, we can easily locate the suspicious statements and figure out the root causes with a bit of manual work. Our experimental data shows that our tool is effective in detecting various resource leaks in Android apps.

Our current research focuses on system resources which can lead to performance degradation and even system crash. The next step of our work is to make quantitative analysis of the performance. And for the features of event-driven programming, there exist many UI-based events in Android apps, extracting UI-based event sequences and using model checking to do more precise analysis may be another direction of future work.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments. This work was partially supported by the National Natural Science Foundation of China under grant No. 61070039.

References

[1] "Activity Lifetime." URL:http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle
[2] "Agenda Widget." URL: http://www.androidagendawidget.com
[3] "Androguard." URL: http://code.google.com/p/androguard/
[4] "Baidu IME." URL:https://play.google.com/store/apps/details?id=com.baidu.input
[5] "Baidu Map." URL: https://play.google.com/store/apps/details?id=com.baidu.BaiduMap
[6] "Baidu Voice Assistant." URL:https://play.google.com/store/apps/details?id=com.baidu.voiceassistant
[7] "Camera API." URL:http://developer.android.com/reference/android/hardware/Camera.html
[8] "Camera resource hasn't been released properly, other applications fail to connect to camera." URL:http://stackoverflow.com/questions/2563973/android-fail-to-connect-to-camera
[9] "Camera resource is unavailable when another application hasn't released it." URL: http://tieba.baidu.com/p/2180732543
[10] "ContentProviderClient API." URL:http://developer.android.com/reference/android/content/ContentProviderClient.html
[11] "Dalvik Opcodes." URL:http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html
[12] "Dare." URL: http://siis.cse.psu.edu/dare/index.html
[13] "ej-technologies GmbH. JProfiler." URL: http://www.ej-technologies.com
[14] "FindBugs." URL:http://findbugs.sourceforge.net
[15] "FindBugs plugin for Android specific coding." URL:https://code.google.com/p/findbugs-for-android
[16] "Gmail application keeps trying to reload or screen flickering." URL: http://forums.androidcentral.com/sprint-photon/162959-gmail-application-keeps-trying-reload-screen-flickering.html
[17] "Gmail app hangs and continuously flickers between app and home-screen." URL:https://code.google.com/p/android/issues/detail?id=27512
[18] "Google Email." URL:https://play.google.com/store/apps/details?id=com.google.android.gm
[19] "JD-GUI." URL: http://java.decompiler.free.fr/?q=jdgui
[20] "K9mail: Simplifying wakelock." URL: http://code.google.com/p/k9mail/source/detail?r=1696
[21] "Klocwork's capabilities for Android device development." URL:http://www.klocwork.com/solutions/android-development/index.php
[22] "Klocwork's inspection for Android specific coding." URL:http://www.klocwork.com/products/documentation/current/Java_checker_reference
[23] "MAT." URL: http://www.eclipse.org/mat
[24] "MediaPlayer API." URL:http://developer.android.com/reference/android/media/MediaPlayer.html
[25] "Osmdroid." URL: http://code.google.com/p/osmdroid
[26] "Osmdroid Issue 53." URL: http://code.google.com/p/osmdroid/issues/detail?id=53
[27] "ProGuard." URL: http://developer.android.com/tools/help/proguard.html
[28] "Quest Software. JProbe." URL: http://www.quest.com/jprobe
[29] "Registered sensor hasn't been released in Baidu Map." URL:http://tieba.baidu.com/p/1364806960
[30] "Registered sensor hasn't been released in Baidu Map." URL:http://tieba.baidu.com/p/1655100103
[31] "Registered sensor hasn't been released in Baidu Map." URL:http://tieba.baidu.com/p/1671759749

[32] "SensorManager API." URL:http://developer.android.com/reference/android/hardware/SensorManager.html
[33] "Soot." URL: http://www.bodden.de/2008/09/22/soot-intra
[34] "WALA." URL: http://wala.sourceforge.net/wiki/index.php/Main_Page
[35] "Youku." URL: https://play.google.com/store/apps/details?id=com.youku.phone&hl=zh
[36] Shuaiqi Liu. Source code materials of "Android mobile application development from beginner to the master" (in Chinese). URL:http://www.tdpress.com/zyzx/tsscflwj/268575.shtml
[37] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, Samuel P. Midkiff. What is keeping my phone awake? Characterizing and Detecting No-Sleep Energy Bugs in Smartphone Apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, Pages 267-280, 2012.
[38] Abhinav Pathak, Y. Charlie Hu, Ming Zhang. Where is the energy spent inside my app? Fine Grained Energy Accounting on Smartphones with Eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, Pages 29-42, 2012.
[39] Damien Octeau, Somesh Jha, Patrick McDaniel. Retargeting Android applications to Java bytecode. In *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20)*, 2012.
[40] E. Torlak and S. Chandra. Effective interprocedural resource leak detection. In *Proceedings of 2010 ACM/IEEE 32nd International Conference on Software Engineering (ICSE 10)*, Pages 535-544, 2010.
[41] G. Michael, Z. Yajin, W. Zhi, and J. Xuxian. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS 12, 2012.
[42] M. Arnold, M. Vechev, and E. Yahav. QVM: an efficient runtime for detecting defects in deployed systems. *ACM Trans. Software Engineering and Methodology*, vol.21, Pages 2:1-2:35, 2011.
[43] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pages 61-72, 2006.
[44] M. Hauswirth and T.M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pages 156-164, 2004.
[45] M. Jump and K. S. McKinley. Cork. Dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Pages 31-38, 2007.
[46] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *17th European Conference on Object-Oriented Programming (ECOOP)*, Pages 351-377, 2003.
[47] Yepang Liu, Chang Xu, and S.C. Cheung. Where Has My Battery Gone? Finding Sensor Related Energy Black Holes in Smartphone Applications. In *Proceedings of the 11th IEEE International Conference on Pervasive Computing and Communications (PERCOM 2013)*, Pages 2-10, 2013.
[48] Wengang Yin, Bin Yang. Memory Leak and Avoiding Method of Android Application Program (in Chinese). *Microcontrollers & Embedded Systems*, 12 (6), 2012.
[49] Zhenbo Xu, Jian Zhang, Zhongxing Xu. Memory Leak Detection Based on Memory State Transition Graph. In *Proceedings of 19th Asia-Pacific Software Engineering Conference (APSEC 2011)* Pages 33-40. 2011.