Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps

Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang, *Senior Member, IEEE*

Abstract—Android devices include many embedded resources such as Camera, Media Player and Sensors. These resources require programmers to explicitly request and release them. Missing release operations might cause serious problems such as performance degradation or system crash. This kind of defects is called *resource leak*. Despite a large body of existing works on testing and analyzing Android apps, there still remain several challenging problems. In this work, we present Relda2, a light-weight and precise static resource leak detection tool. We first systematically collected a resource table, which includes the resources that the Android reference requires developers release manually. Based on this table, we designed a general approach to automatically detect resource leaks. To make a more precise inter-procedural analysis, we construct a Function Call Graph for each Android application, which handles function calls of user-defined methods and the callbacks invoked by the Android framework at the same time. To evaluate Relda2's effectiveness and practical applicability, we downloaded 103 apps from popular app stores and an open source community, and found 67 real resource leaks, which we have confirmed manually.

Index Terms—Android apps, resource leak, static analysis, byte-code analysis, inter-procedural analysis

1 Introduction

Android smartphones are becoming increasingly popular. A recent report shows that Android's share reaches 82.8 percent in smartphone markets [2]. However, the quality of Android apps is still worrisome, since a majority of Android apps are developed by relatively small teams, which may not afford extensive and expensive testing.

To enrich user experience, Android phones come with many components embedded in them. These components can be divided into two categories: *traditional resources* that can be found in desktop, like CPU, memory and screen, and *exotic resources*, such as GPS, Camera and different kinds of Sensors. Most of the exotic resources are the biggest energy consumers in Android phones, and they drain phone's battery at a high rate [3], [4], [5]. In addition, these resources require explicit user management, that is, the developers need to request and release them manually. Absence of their release operations may lead to huge energy-consumption, memory consumption, or even system crash. We call this kind of defects *resource leaks*.

Unfortunately, resource management is a challenging task for developers. There are a number of reasons for that. First, unlike memory which is allocated and recycled by the virtual machine, these resources require programmers to explicitly turn them on and off. It is challenging for developers to manage the resources correctly, since the execution paths of an Android app are often complicated due to Android's event-driven nature and the large number of callbacks. Even an experienced programmer may forget to release the resources along some possible event sequences. Second, developers often have to push their apps out to markets in a short time, and they tend to focus on the user friendliness and functionality of their apps. Therefore, developers often overlook complaints of performance related problems from users, especially when these complaints contain insufficient information for localizing the bugs. Third, programmers may misunderstand the Android Application Programming Interface (API) specifications. For example, a recent work [6] shows that Android APIs are evolving at the rate of 115 API updates per month on average. The continuous upgrades of Android Software Development Kit (SDK) increase the difficulty of understanding API contracts.

Some existing studies have focused on testing Android apps [7], [8], [9], [10], [11], [12]. However, the testing effectiveness depends on the fault detection capability of test suites. On the other hand, program analysis [13], [14], [15], [16], [17] can detect incorrect behavior and performance degradation of Android apps without executing test cases. However, previous works on program analysis for Android apps almost focus on memory leak, privacy leak or energy bugs, rather than resource leak.

To sum up, resource leak is a common type of bugs in Android apps, but developers often do not have enough time to detect and fix them. Informally speaking, detecting a resource leak in the app is to find a reachable program path that requests but not releases the resource. In this paper, our

- *T. Wu, J. Liu, and J. Zhang are with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and also with University of Chinese Academy of Sciences. E-mail: {wuty, liujr, xuzb, guocr, zhangyl, zj}@ios.ac.cn.*
- *J. Yan is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and also with the Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences. E-mail: yanjun@ios.ac.cn.*

Manuscript received 13 Jan. 2015; revised 1 Feb. 2016; accepted 6 Mar. 2016. Date of publication 27 Mar. 2016; date of current version 18 Nov. 2016. Recommended for acceptance by T. Bultan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2547385

goal is to design a system that can detect resource leaks in Android apps automatically. It can be used for analyzing the apps before they are deployed into the markets. To make this system practically applicable, we summarize the following four crucial considerations.

- *Resource identification.* The first step for detecting resource leaks is to identify target resources. In this paper, we focus on three kinds of resources: *exclusive*, *memory consuming* and *energy consuming* resources (see Section 2.4). They should be released explicitly in an appropriate position as specified in the Android Reference [18].
- *Implicit callbacks.* Android apps are driven by events and callbacks. For example, *onDestroy*() is a callback in each activity (see Section 2.2) of Android apps, which is called when the activity is going to be finished. Programmers often release the resources used by their apps in this method. However, it is invoked by the Android framework and this call relation can not be observed by scanning the code of apps directly. Without considering these implicit callbacks, we may get extra false positives or false negatives.
- *Target object.* Many static analysis tools focus on source code. But we think analysis on byte-code is a reasonable choice for Android apps. The source code of apps is often unavailable for third-party testers, like managers of app markets. Although there are several well-known decompilation tools (e.g., dex2-jar [19]), they do not work well on arbitrary apps. On the other hand, byte-code is closer to the executable software. Attackers can modify the byte-code of an app using unofficial compilers to change the original behaviour of the app. Therefore we choose the byte-code instead of source code as our analysis target.
- *Practicability.* The functionalities of Android apps are becoming richer and more complex, and the size of a typical state-of-art APK file has reached to dozens of megabytes. In addition, a large number of apps are uploaded to Android markets every day. Complete and precise inter-procedural static analysis will take a long time to analyze such large-scale apps. On the other hand, the imprecise analysis methods will lead to many false positives, which increase bug confirmation time for developers. To make our system practical for developers and testers, we take a trade-off between analysis precision and time cost.

For the first issue, we collect a resource table after examining the classes related to the resources from the Android Reference [18]. This table includes the resources that the Reference declares to be released manually. For the second issue, we also manually extract the implicit callbacks from the Android Reference to construct a Callback Graph (CBG), which we consider fundamental for our analysis, such as lifecycle callbacks and user-triggered callbacks.

Byte-code plays an important role in the Android system. Although Android apps are mostly implemented in Java, they are compiled to the Dalvik byte-codes (instead of Java byte-codes) and packed in a Dalvik EXecutable (DEX) file to be deployed. In the former versions of Android 4.4, the Dalvik byte-codes are run on Dalvik Virtual Machine (DVM), and they are compiled to the local machine codes in Android 4.4 and later versions. However, most of existing static analysis tools (e.g., [15], [16]) for Android apps do not analyze the Dalvik byte-codes directly, and they are built on top of static analysis tools (like Soot, WALA) for Java programs. To analyze the Android apps, these tools have to translate the Dalvik byte-codes to some intermediate representation (like Jimple) or Java byte-codes. Several works have mentioned that the translation process may fail in some cases [20], [21], especially in the deliberately modified apps. The above issues motivate us to construct a pure Dalvik-byte-code-oriented analysis tool for Android apps. To analyze the Dalvik byte-codes, we employ an open-source tool called `Androguard` [22] to parse DEX files and generate the Control Flow Graph (CFGs) of apps for further analysis.

For the last issue, we choose several light-weight techniques in our static analysis approach. According to our experience, these techniques are good enough for resource leak detection. Our observation is that the number of resource leaks reported by our approach for a specific app is limited (See Section 8.5). Therefore, the cost of bug confirmation is usually acceptable for developers. Some extremely precise techniques (e.g., path-sensitive analysis) are not adopted in our approach, for they will lead to a sharp increase in execution time for analyzing large-scale Android apps, that may be unacceptable for frequently upgrading apps.

Based on these thoughts, we propose a light-weight and inter-procedural approach to detect resource leaks in Android apps. This approach is based on Function Call Graph (FCG) analysis, which handles the features of the callbacks defined in the Android framework. To adapt to different user requirements, our approach contains two analysis techniques with different analysis precision, including flow-insensitive and flow-sensitive. The flow-insensitive analysis has been proposed in our previous work [1]. It scans byte-codes sequentially and records resource request and release operations at each method in the apps. Then it detects resource leaks through matching these operations. Without considering the control flow information, this technique can quickly analyze an app, but it will lead to some false negatives. We can use the flow-sensitive technique to improve our analysis with fewer false negatives. We first construct all the CFGs of an Android app by `Androguard`. Then, we transform each CFG into a more concise model called Value Flow Graph (VFG), which only reserves the control flow and resource-related information, to reduce the scale of the analysis model. Finally we check the VFG model to find the resource leaks.

The contributions of this paper is four-fold. First, we systematically define the problem of resource leaks in Android apps. We collect a table including the resources that the Android Reference declares to be released manually. To the best of our knowledge, we are the first ones focusing on this problem. Second, we design a general approach to detect resource leaks in Android apps automatically and propose two analysis techniques (flow-insensitive and flow-sensitive) to adapt to different requirements of detection efficiency and accuracy. Third, we develop a light-weight and inter-procedural static analysis tool called Relda2 to analyze an app's resource operations and locate resource leaks automatically. Finally, to evaluate the effectiveness of Relda2, we perform an extensive experimental evaluation with 103 real-world apps,

```
<manifest android:versionName="1.0">
<application>
   <activity android:name="MainActivity">
      <intent-filter>
         <action android:name="android.intent.
              action.MAIN"/>
         <category android:name="android.intent.
              category.LAUNCHER"/>
      </intent-filter>
   </activity>
</application>
</manifest>
```

Fig. 1. A portion of a manifest file.

which are downloaded from two famous app markets [23], [24] and an open source community [25]. In our experimental results, our tool reports that about 74 percent of the apps have resource leaks, among which we have confirmed 67 real resource leaks in 37 apps via manual code review (if we have source code) and error-guessing testing [26].

The rest of the paper is organized as follows. We begin with background in the next section. In Section 3, we give an overview of our approach. In the following four sections, we present our resource leak detection techniques in detail and illustrate how they work using a simple example. Experimental results are shown in Section 8. Section 9 discusses the related works. We give the conclusions and discuss future work in the last section.

2 Background

This section provides the background knowledge necessary for further discussion, mainly about the Android framework and the resource leak problem.

2.1 Android Basics

Android apps are composed of four types of components: Activities (which provide an interface with which users can interact), Services (which perform long-running operations in the background without interaction with the user), Content Providers (which manage access to a structured data set such as database) and Broadcast Receivers (which react to broadcast messages). Unlike conventional Java programs, Android apps do not have a single entry point such as function *main*(). Instead, an app contains one or more components defined in its manifest file. For example, Fig. 1 shows a part of the manifest file of an Android app. It defines the activity *MainActivity* as the entry component of this app.

2.2 Activity Lifecycle

Since the mechanism of the Service component is similar to that of the Activity component, and the mechanisms of the Content Provider and Broadcast Receiver are relatively simpler, we only introduce the Activity component for brevity. In an Android app, each activity is required to follow a lifecycle defining how it should be created, used, and destroyed. There are several callbacks in the lifecycle (shown in Fig. 2), where developers can override them to handle state transitions in the lifecycle. Specifically, an activity starts with *onCreate*() and lasts until *onDestroy*() is called. The loop defined by *onStart*() and *onStop*() is the visible lifetime of the app. An activity's foreground lifetime

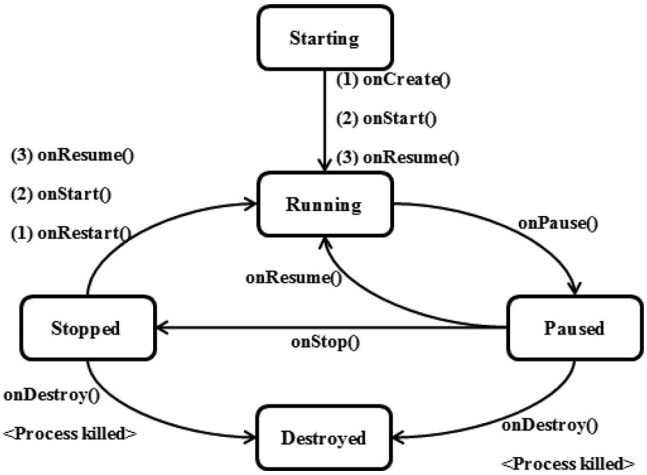


Fig. 2. Activity lifecycle of an Android app.

(i.e., the "Running" state), in which the activity can interact with users, starts with a call to *onResume*() and ends with a call to *onPause*().

2.3 Implicit Callbacks

Similar to Java Graphical User Interface (GUI) programs, Android apps are usually driven by events and callbacks. We can divide the callbacks into two categories: system-triggered and user-triggered.

A system-triggered callback is activated automatically by the Android framework, when the app reaches a certain state or invokes a specific method. For instance, the callbacks in the lifecycle of an activity are all system-triggered callbacks. According to our studies, there are lots of similar system-triggered callbacks defined in the Android framework, which are related to resources. Taking the Camera resource as an example, a series of callbacks are provided for assisting it to capture images. Among them, the *shutter*() callback is invoked after the image is captured and the *raw*() callback occurs when the raw image data is available.

The other callbacks are triggered by user events to handle user interactions with GUI components. We call this kind of callbacks user-triggered callbacks. In general, these callbacks contain click and screen-touch callbacks (e.g., *onClick*(), *onLongClick*(), *onTouch*()), keyboard callbacks (e.g., *onKeyUp*(), *onKeyDown*()), state change callbacks (e.g., *onFocusChanged*(), *onItemSelected*()), and so on.

The callbacks are invoked by the Android framework and their call relations are implicit in the byte-codes of the apps. They will be just regarded as unreachable methods, if we scan the byte-codes without further analysis. How to analyze these implicit calls can be seen in Section 4.2.

2.4 Resource Leak

In preparation for our study, we collected some resource leak instances in Android apps from several famous mobile app forums and discussion groups in China. We found that many users complained about problems of performance degradation, energy drain and application crashes, most of which are due to the inappropriate use of the exotic resources [27], [28], like Camera and Sensor components. Then we checked the Android reference about these components, and categorized the resources into the following three classes.

```
public class MainActivity extends Activity {
   protected void onCreate(Bundle
                  savedInstanceState) {
1.    Intent cameraIntent = new Intent
      (MainActivity.this, CameraActivity.class);
2.    startActivity(cameraIntent);
      ......
   }
}

public class CameraActivity extends Activity {
   protected void onCreate(Bundle
                  savedInstanceState) {
3.    takePicture();
      ......
   }
   protected void onPause() {
4.    release();
      ......
   }
   private void takePicture() {
5.    mCamera = Camera.open();
6.    mCamera.startPreview();
   }
   private void release() {
7.    if(bIfPreview) {
         ......
      }
8.    else {
         ......
      }
9.    if(mAllDoFlag) {
10.      mCamera.stopPreview();
11.      mCamera.release();
      } else {
         ......
      }
   }
   protected void onStart() {
      ......
12.   mCamera.startFaceDetection();
      ......
   }
}
```

Fig. 3. Example code containing a resource leak.

- *Exclusive resources* can only be used by one app at a time. Failing to release these resources will prevent other apps from accessing them. For example, the Camera component is an exclusive resource. The Android reference says [29]: "*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()).*"
- *Memory consuming resources* consume much more memory than general resources. For example, the Media Player component is one of these resources. The Android reference says [30]: "*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*".
- *Energy consuming resources* consume much more energy than general resources. For instance, the sensor components are one kind of these resources. To help users to access the device's sensors, the Android platform provides a class called SensorManager and a service called SENSOR_SERVICE. When users want to employ some sensor, they need to register it as a SensorManager object to SENSOR_SERVICE, and should cancel the registration for the SensorManager objects when they do not need it anymore. The Android reference says about Sensor [31]: "*Always make sure to disable sensors you do not 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*".

Example 1. Consider the code snippet from an Android app in Fig. 3, which uses the *Camera* resource. It contains two Activities, *MainActivity* and *CameraActivity*. As soon as the app is launched, *MainActivity* is activated and it starts *CameraActivity* with the API *startActivity* (Line 2). The methods *Camera.open*() and *mCamera.startPreview*() are two Android APIs to request the *Camera* resource, and the methods *mCamera.stopPreview*() and *mCamera.release*() are two APIs to release the *Camera* resource. We can easily discover that there exists a resource leak in this program. Specifically, in the method *release*(), if the variable *mAllDoFlag* is *false*, then *mCamera.stopPreview* and *mCamera.release*() will not be executed, thus the resource *Camera* is leaked. As *Camera* is an exclusive resource, other apps will not be able to use it until this app is killed.

3 Overall Architecture

Given an Android app, the goal of this work is to detect whether the app has resource leaks automatically. As mentioned above, the efficiency and scalability have become the bottleneck of static analysis on Android apps. To deal with these problems, our approach should possess the following characteristics:

- *Full automation*. Testers only need to provide target APKs and our approach can work directly without any extra user interactions.
- *High efficiency with enough accuracy*. Typically, static analysis of real-world and large-scale Android apps is time-consuming. Moreover, bug confirmation and fixing are also labor-intensive for developers. Therefore, our approach must process the apps efficiently with acceptable number of false positives.
- *Scalability*. The Android framework is always upgrading and will add new resources. Thus we collect the resource request and release APIs as a static file to describe the analysis objectives. This file can adapt to the possible changes of the target resources provided by Android SDK, and the users can modify the file to customize the analysis.

Fig. 4 shows a high-level overview of our approach. It consists of several key parts as follows:

- *Resource and implicit callback identification*. According to our categories of the exotic resources, we collect a resource table containing the resource names, operations and the suggested places to release them. Besides, we also extract the implicit callbacks from the Android framework as a Callback Graph (CBG for short, see the next section).
- *Preprocess*. Our approach accepts an Android APK as input. First, we disassembled the app into DEX byte-codes. Then we traverse the byte-codes in a

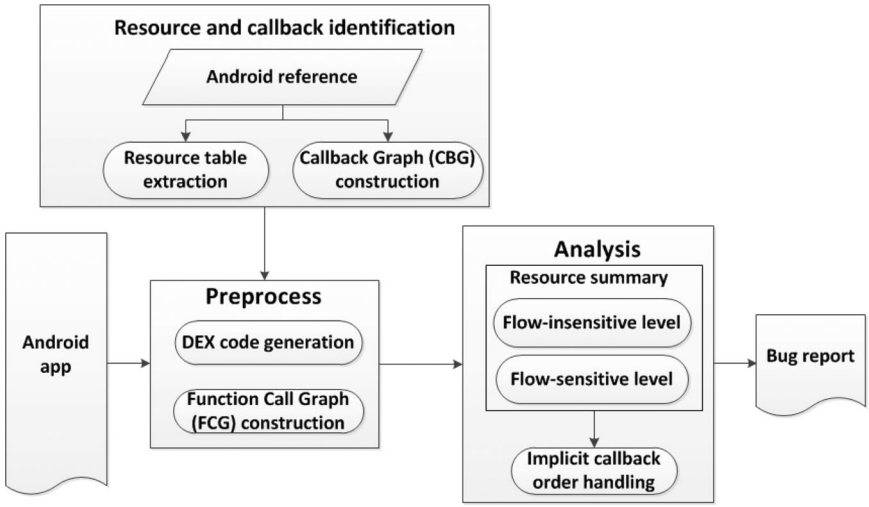


Fig. 4. High-level overview of our approach.

sequential order to construct a Function Call Graph, which contains the call relations of user-defined methods and implicit callbacks. FCG and CBG are the basis for inter-procedural analysis.

- *Analysis*. With the FCG of the app, we can systematically traverse the methods in the apps, and trace the resource request and release operations. Our analysis approach can be divided into the following two stages.
  - *Resource summary*. To perform inter-procedural analysis, a straightforward approach is to inline each invoked function. Obviously, this technique can precisely capture the effects of invoked functions. However, it may result in path explosion, and is hard to be applied to large-scale apps. Function summary is an alternative technique to abstract the codes of callee functions. Based on Android characteristics and our analysis requirements, we define a method's summary (called resource summary) as a set, which summarizes the resource request and release operations used in this method. Depending on the analysis precision, the resource summary can be generated with or without the control flow information in the method.
  - *Implicit callback order handling*. Several implicit callbacks have execution orders. Recall the callbacks in lifecycle (Fig. 2), when an activity is first activated, the framework first invokes the method *onCreate*(), and then invokes the methods *onStart*() and *onResume*() in order after the method *onCreate*() finishes. These orders are not included in FCGs, for FCGs only represent the call relationships between methods. Lacking the callback order information leads to inaccurate results. We traverse the CBG to get the implicit callback orders and refine the resource summaries.
- *Bug report*. Finally, we check the resource summary of each method to get bug reports. If there are resource leaks in the app, we can report the location (like class name or method name) where these leaks occur and some trace information to help developers to locate them.

We explain the above steps in detail in the following sections. Specifically, Section 4 introduces resource-related operation extraction and callback graph construction, while Section 5 shows the detailed process of FCG construction. Two kinds of resource summaries, including flow-insensitive and flow-sensitive, are given in Sections 6 and 7, respectively.

For convenience, we list some acronyms that are frequently used in this paper:

- API (Application Programming Interface)
- CBG (Callback Graph)
- CC (Cyclomatic Complexity)
- CFG (Control Flow Graph)
- CG (Call Graph)
- CTL (Computation Tree Logic)
- FCG (Function Call Graph)
- GUI (Graphical User Interface)
- LKS (Labeled Kripke Structure)
- SDK (Software Development Kit)
- VFG (Value Flow Graph)

4 Resource-Related Operation Identification and Callback Graph Construction

In this section, we more fully describe the details of resource related operation identification and callback graph construction. Currently, we perform these two steps manually according to the Android reference [18]. Note that these steps only need to be done once, and have been embedded in the tool Relda2.

4.1 Resource-Related Operation Identification

To identify target resources, we searched the Android Reference, using several keywords (shown in Table 1), which may be related to the methods for requesting and releasing resources.

We also extended them with regular expressions to obtain more related keywords. For example, the keyword "start" is modified to the style "start*", which means matching any member method whose name starts with "start", e.g.,

TABLE 1
Keywords Matching Resource Request and Release Methods of Classes

add	begin	create	construct	enable
load	lock	mount	obtain	open
new	register	request	require	start
abandon	cancel	clear	close	end
disable	finish	recycle	release	remove
stop	unload	unlock	unmount	unregister

"startPreview" in the class *Camera* is one such case. We scanned the summary section of the methods from 1944 classes in the manual with the extended keywords. Then, we got 313 methods whose names match one of the keywords. We manually checked their detailed descriptions to determine whether the method is related to the resources. If it is used to request some resource, then we figured out whether the resource should be released manually. According to the Android Reference, the resources should be released in three callbacks including *onPause*(), *onStop*(), and *onDestroy*() that we call the *exit callbacks*. We read through the training sections of the resources and identified the "acquire" and "release" callback methods that should be called by application components. We found 64 methods of 45 resources that are involved when releasing these resources manually. Some frequently used resource request and release operations are summarized in Table 2. The whole resource table can be seen in our web site http://lcs.ios.ac.cn/~zj/ResourceTable.html.

4.2 Callback Graph Construction

To expose all the implicit callback relations in Android framework, we construct a Callback Graph that consists of several separate trees. Each tree represents the sequences of implicit callbacks of a component, a class or a method, where each chain of a tree from the root to a leaf describes an order in which the callbacks in the chain are triggered by the Android framework. In the CBG, we mainly construct the trees for the four commonly used components, including Activity, Service, Broadcast Receiver, and Content Provider. Apart from them, we also consider several asynchronous callbacks, such as *AsyncTask* and *handleMessage*.

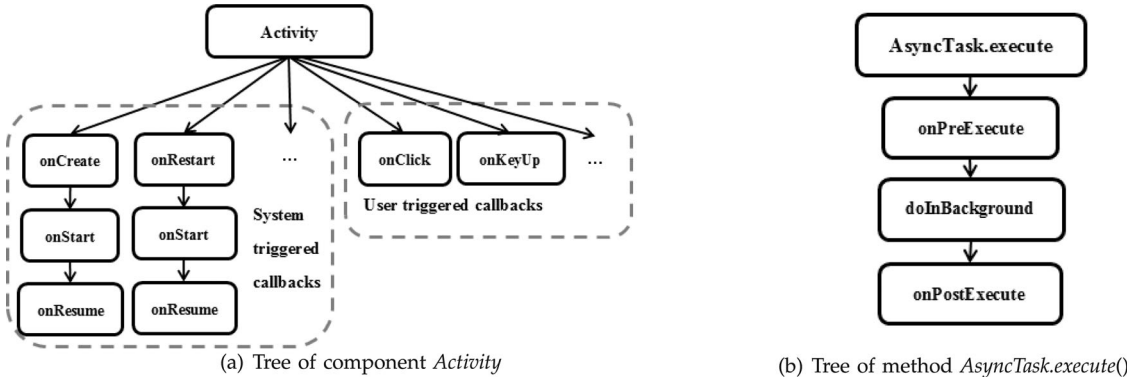
Fig. 5 gives two trees in the CBG. Fig. 5a shows the tree of the component *Activity* including several system-triggered callbacks (*onCreate*(), *onPause*(), ...) and user-triggered callbacks (*onClick*(), *onKeyUp*(), ...). This tree is constructed based on the Activity lifecycle (Fig. 2). We extract all the possible simple paths (each callback occurs at most once) from the lifecycle, and insert the paths into the tree. Fig. 5b gives another example representing the tree of the method *AsyncTask.execute*(). The tree contains three callbacks: *onPreExecute*(), *doInBackground*() and *onPostExecute*(). When the method *AsyncTask.execute*() is invoked, these callbacks will be called in order.

5 FCG Construction

In the analysis of an Android app, we distinguish between local resources and global resources. For a local resource (resource declared and only used in a method), we assume

TABLE 2
Summary of Frequently-Used Resource Request and Release Operations

Resource package	Resource name	Operations on resource	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()



(a) Tree of component *Activity* (b) Tree of method *AsyncTask.execute*()

Fig. 5. Parts of CBG.

developers want to release it when the method finishes. To detect the possible leaks, we can analyze each function separately if Android apps only use local resources. However, in real-world Android apps, the resources are usually requested in one function and released in another function. We call this kind of resources global resources that should be released in all the paths that go through its request point, or in the exit callbacks of the app. To address this issue, we construct the Function Call Graph (FCG) to assist inter-procedural analysis.

Here, an FCG is similar to a Call Graph (CG) [32]. It is a directed graph, where each node denotes a method and each edge (f, g) denotes method f calls method g. There are several slight differences between call graph and our FCG. Since Android programs are event-driven and have multiple entry methods, an FCG has multiple entry nodes. In addition, the FCG needs to handle the implicit callbacks defined in the Android framework. Note that, the FCG just represents the calling relationships between methods in the program, while it does not contain the execution order of methods defined by the Android framework which can be found in CBG. For instance, recall the Activity lifecycle, when an activity is activated, the framework first invokes method *onCreate*(), and invokes method *onStart*() after method *onCreate*() finishes. However, in our FCG, there is no edge from method *onCreate*() to method *onStart*() since the former does not actually invoke the latter in itself. The orders will be considered in the subsequent analysis process.

The following gives our FCG construction process. As mentioned in the previous section, an Android app has no entry function (like *main*()). Instead, an Android app appoints the entry point in an XML file named *manifest.xml*. So the first step of FCG construction is to parse the XML file to get the entry activity. We use an open-source tool named `Androguard` to extract these implicit information and generate standard Dalvik byte-codes of an app. Then we start from the entry activity and traverse the byte-codes in a sequential order to obtain a complete FCG. It is trivial to identify the function calls in the byte-codes according to "invoke" instructions.

Algorithm 1 shows the FCG construction algorithm. It takes an Android app (variable *app*) as input. Method *getMainActivity* obtains the main activity of *app*. Then the algorithm starts from the main activity to obtain all the reachable methods and construct nodes and edges for them in the FCG (variable *fcg*). Each node in *fcg* contains an attribute called *children* to represent the corresponding nodes of its invoked functions. For each function, we traverse its instructions one by one. When it comes to an "invoke" instruction (variable *ins*), the algorithm collects the *actual* invoked functions of *ins*. Specifically, if *ins* invokes a callback-irrelevant function, which is not related to the callbacks invoked by Android framework, then the invoked function name can be extracted from this instruction directly. Otherwise, if *ins* invokes a callback-related function, invoked functions are collected according to the callback graph. Then it creates FCG nodes for these invoked functions and adds these new FCG nodes into *children* of their parent node. Finally, it recursively constructs FCG nodes for their invoked functions.

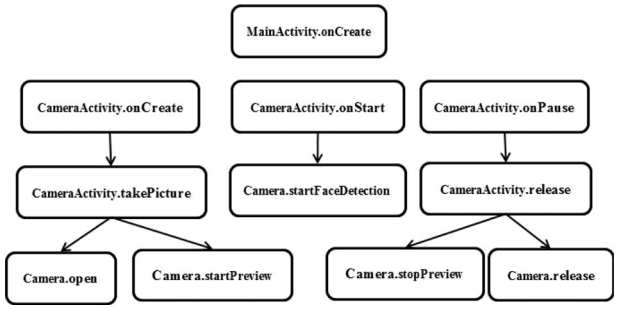


Fig. 6. An example FCG.

Algorithm 1. Construct the FCG

```
1: mainAct = getMainActivity(app)
2: for each invoked function f of mainAct do
3:    create a new FCG node fnode for f in fcg
4:    construct_fcg_nodes(f, fnode)
5: end for
```

Example 2. Consider the code in Example 1 again. It includes two Activities *MainActivity* and *CameraActivity*, where *MainActivity* is the entry Activity and *CameraActivity* is a resource-related Activity. With the tree of the Activity component, we create an FCG node for the method *MainActivity.onCreate*(). Then we find *CameraActivity* is activated in the method *MainActivity.onCreate*() (Line 2 in Fig. 3), and subsequently the method *CameraActivity.onCreate*(), *CameraActivity.onStart*() and *CameraActivity.onPause*() can be triggered. We connect all these calling relations and construct the FCG shown in Fig. 6, where each round rectangle represents a node that corresponds to a function, and each arrow represents a function call.

In our previous work [1], we also described an FCG construction algorithm. However, without CBG, we have to assume all the entry points are reachable, while some of them are useless codes (see Section 8.3). As a result, the experiment of our previous work includes several false positives, which have been eliminated in Relda2.

construct_fcg_nodes(*f*, *fnode*)

```
 1: if f has been visited then
 2:    return
 3: end if
 4: for each instruction ins in f do
 5:    if ins is an "invoke" instruction then
 6:       for each invoked function f' of ins do
 7:          create a new FCG node fnode' for f' in fcg
 8:          if f' is not an implicit callback then
 9:             add fnode' into fnode.children
10:          end if
11:          construct_fcg_nodes(f', fnode')
12:       end for
13:    end if
14: end for
```

6 Flow-Insensitive Analysis

Through the above process, we can build the FCG of an app, which contains all the reachable functions from the

entry activity. The next step is to analyze these functions to detect whether there are resource leaks. We can perform a simple and fast checking by scanning byte-codes in a sequential order without considering the control flow of each method, and thus it is flow-insensitive. We define the summary of a function as a set of resource operations invoked by it directly or indirectly. We traverse all the functions in the FCG in a bottom-up manner and generate their resource summaries. Each recursive function or loop is just analyzed once. At last, we get the resource leaks through observing the unmatched resource request and release operations in the resource summaries.

6.1 Resource Summary

The detailed process of resource summary for each method is shown in Algorithm 2. Given an app and its FCG (the variable *fcg*), it returns resource summaries (the variable *summ_map*) of the functions in *fcg*. Function *bot_to_up()* rearranges items in *fcg* in a bottom-up order and gets *func_list*. This function mainly uses the topological ordering algorithm. Then, for each function (the variable *f*) in *func_list*, we traverse its instructions one by one. Variable *ops* is a set, which is used to store all the request or release operations in the function, and it is initialized as an empty set. If the instruction (the variable *ins*) is related to resources, that is, *ins* invokes a request or release operation about some resource, then we use function *getOp* to extract the name of this operation from *ins* and add it to *ops*. Apart from these direct instructions, if the instruction invokes function f', then we add the summary of f' (*summ_map*[f'], that may include the resource-related operations invoked indirectly by *f*) to *ops*. Function *summary* simplifies the set *ops* by eliminating the matched resource operations and generates the ultimate summary of function *f*. For example, suppose that $ops = \{a_1, a_2, b_1, a_3, b_2\}$, where a_i and b_i indicate the request and release operations for the ith resource. Then we eliminate (a_1, b_1) and (a_2, b_2), and get the resource summary $\{a_3\}$.

Algorithm 2. Obtain Resource Summaries of each Function in the FCG

```
1: summ_map = { }
2: func_list = bot_to_up(fcg)
3: for each function f in func_list do
4:    ops = ∅
5:    for each instruction ins in f do
6:       if ins is related to resources then
7:       ops = ops ∪ (getOp(ins))
8:    else if ins invokes function f' then
9:       ops = ops ∪ (summ_map[f'])
10:   end if
11:  end for
12:   summ_map[f] = summary(ops)
13: end for
```

6.2 Implicit Callback Order

Until now, our analysis has not considered the sequence of the callback methods of a component or class in the application. We now use the CBG to combine and reduce resource summaries for methods in the same sequence/chain in the CBG. For each chain (which corresponds to an implicit callback order) in the CBG, we figure out its all possible execution sequences in the app. Then for each sequence, we sum up the resource summaries of all the methods in the sequence as the summary of the first method in the sequence. The summary rearrangement algorithm is similar to Algorithm 2.

After that, we get a complete resource summary for each method in the app. When the summary of a method f still contains a request operation about the resource r, we check whether its corresponding exit callback(s) contain the release operation. If there is no such operation, it indicates that the method f requests the resource r without releasing it , i.e., the function *f* has a resource leak.

Example 3. We use an example to illustrate how the above process works. Consider the CBG chain *onCreate*()→*onStart*()→*onResume*() and the app in Example 1, we first get the execution sequence *CameraActivity.onCreate*()→*CameraActivity.onStart*(). The resource summaries of these two methods calculated by Algorithm 3 are {*Camera.open*, *Camera.startPreview*} and {*Camera.startFaceDetection*}. We sum up these two summaries and get the new resource summary of the method *CameraActivity.onCreate*(): {*Camera.open*, *Camera.startPreview*, *Camera.startFaceDetection*}. As the summary still has three resource request operations, we check the suggested exit callback *CameraActivity.onPause*(), whose resource summary is {*Camera.stopPreview*, *Camera.release*}. We find that this callback has the release operations for *Camera.open* and *Camera.startPreview*, while it does not contain that for *Camera.startFaceDetection*. Therefore, we report there is a resource leak in the method *CameraActivity.onCreate*().

6.3 Discussion

We will miss several resource leaks with the flow-insensitive analysis, as we do not consider the control flow information. Recall Example 1, the resource summary of the method *release*() is [$Camera.stopPreview$, $Camera.release$], which denotes that we consider the method *release*() invokes these two release operations in the method. However, there are several program paths in this method that do not release the resource (when the branch in Line 9 takes the false side). That is, the flow-insensitive technique may have false negatives.

7 Flow-Sensitive Analysis

In this section, we introduce another detection technique which is flow-sensitive. In general, flow-sensitive static analysis is based on CFGs. However, the number of CFGs in an Android app is often large (it may reach hundreds of thousands), and the structure of CFGs is becoming more and more complex. So trivial analysis on them will be too inefficient to be applied to large-scale apps. We observe that there is often only a small part of the whole app which is related to resources. Therefore, we can prune the app into a more concise model that only reserves the necessary control flow and resource-related information. The concise model used here is modified from Value Flow Graph [33], [34], [35]. With the VFGs, we can apply the conventional static

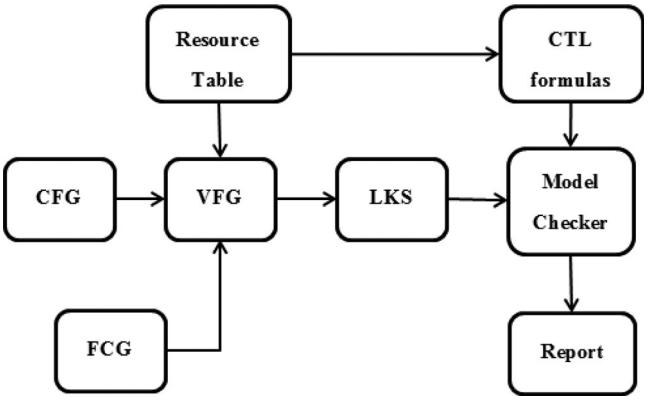


Fig. 7. Overview of flow-sensitive analysis.

analysis techniques (some graph-searching and path-reachability algorithms) to detect the resource leaks. However, we find that the property of resource leak can be easily described with Computation Tree Logic (CTL), and the detection problem can be treated as a model checking problem. Furthermore, the nature of the pure static analysis and model checking techniques are both based on several searching algorithms, and many model checking tools have been implemented with several elaborate optimizations to prune the search space and accelerate the efficiency. Thus we choose model checking as the main detection technique.

Fig. 7 shows a high-level overview of our flow-sensitive detection technique. First, we employ `Androguard` to extract the CFG of each method in the target app. Then we propose an algorithm to reduce the CFGs and obtain the corresponding VFGs automatically. For each VFG, we translate it into an LKS (a model used in model checking, see Section 7.1.2) and use several CTL formulas to describe the resource related properties. Then we model check the LKS to construct the resource summary of the method.

Like the flow-insensitive technique, we perform the inter-procedural analysis based on the resource summary. The summary for each method is different from that used in the flow-insensitive technique, since we take into account the control flow information. Here the summary of a method is divided into two disjoint subsets as follows.

- *Unreleased resource set*. It consists of several resource request operations. Each operation in the set satisfies the following property: there exists at least one path in the method such that the resource is requested in this path but not released.
- *Unrequested resource set*. It consists of several resource release operations. Each operation in the set satisfies the following property: for any path in the method, the resource is released in this path and there is not any corresponding request operation located before this release operation.

After constructing the resource summaries of each method, we need to deal with the implicit callbacks. The process is similar to that in the flow-insensitive technique.

Before showing the details of the detection technique, we will introduce several notations about VFG and model checking.

7.1 VFG and Symbolic Model Checking for CTL

7.1.1 The Notion of VFG

As discussed above, VFG nodes can be classified into two categories: control flow related nodes and resource related nodes. Specifically, control flow related nodes consist of the following three kinds:

- *EntryNode* The unique entry node of a VFG;
- *CallNode* Function call node. An "invoke" instruction corresponds to a CallNode;
- *ExitNode* Function return node. A "return" instruction corresponds to an ExitNode.

The resource related nodes can be defined as follows:

- *SourceNode* Resource request node. A resource request instruction corresponds to a SourceNode;
- *FreeNode* Resource release node. A resource release instruction corresponds to a FreeNode.

As a matter of convenience, we call the kinds of instructions mentioned above ("invoke" instructions, "return" instructions, resource request instructions and resource release instructions) *VFG-related instructions*.

7.1.2 CTL and Symbolic Model Checking

Computation tree logic [36] is a branching-time logic, which means that its model of time is a tree-like structure. It is used in formal verification of software or hardware artifacts, typically by symbolic model checking [37]. Let AP denote the underlying set of atomic propositions. The CTL formulas are defined by the following grammar:

$$\begin{aligned}\phi ::= & \ p|\neg\phi|\phi \wedge \phi|\phi \vee \phi|\phi \rightarrow \phi \\ & EX\phi|EF\phi|E(\phi U\phi)|E(\phi R\phi) \\ & AX\phi|AF\phi|AG\phi|A(\phi U\phi)|A(\phi R\phi),\end{aligned}$$

where $p \in AP$ and ϕ is a CTL formula.

The semantics of CTL is defined with respect to a *Labeled Kripke Structure* (LKS) $M = \langle S, \Delta, I, L\rangle$, where S is a finite set of states; $\Delta \subseteq S \times S$ is the set of transitions; $I \subseteq S$ is the set of initial states; and $L : S \rightarrow 2^{AP}$ assigns each state a set of atomic propositions, whose elements hold in that state.

We use $M, s \models \phi$ to denote that formula ϕ holds at state s in structure M. The relation $\models$ for AG and AF are defined as follows and the paper [36] gives more details.

$M, s \models AG(\phi)$ means that ϕ holds at every state in every path from s.

$M, s \models AF(\phi)$ means that ϕ holds in the future along every path from s.

Given a system model and a CTL formula, the goal of symbolic model checking is to provide either a claim that the formula is satisfied in the model or a counter-example falsifying the formula.

7.2 VFG Construction

The VFG construction algorithm is summarized by Algorithm 3. It takes a CFG (the variable *cfg*) as input and generates the corresponding VFG (the variable *vfg*). The variable *node_map* matches the CFG blocks and the VFG nodes. The algorithm first creates a RootNode (the variable *root*) as the

entry node of *vfg*. Then it creates the VFG nodes and edges in the following two for-loops, respectively.

Algorithm 3. Construct VFGs

```
1: node_map = {}
2: create a RootNode root as the entry node of vfg
3: for each block b in cfg do
4:    (firstnode, lastnode) = create_vfg_nodes(b)
5:    node_map[b] = (firstnode, lastnode)
6:    if b is the entry block of cfg then
7:       create an edge from root to firstnode
8:    end if
9: end for
10: for each block b in cfg do
11:    (firstnode, lastnode) = node_map[b]
12:    for each block b' in b.nexts do
13:       (firstnode', lastnode') = node_map[b']
14:       create an edge from lastnode to firstnode'
15:    end for
16: end for
17: minimize vfg
```

In the first for-loop (Line 3-9), it traverses the CFG blocks one by one and creates corresponding VFG nodes for each block (the variable *b*) via the function *create_vfg_nodes*. Note that the corresponding VFG nodes of each CFG block are a chain of nodes rather than a single node, since a CFG block may contain multiple VFG-related instructions, each of which corresponds to a VFG node. Function *create_vfg_nodes* returns the first and last nodes (the variables *firstnode* and *lastnode*) of the chain of the VFG nodes corresponding to the CFG block *b*.

In the second for-loop (Line 10-16), it traverses the CFG blocks again to create VFG edges. Variable *b.nexts* denotes all the adjacent blocks of block *b*. For each adjacent block (the variable *b'*), it creates an edge from the last VFG node of block *b* (the variable *lastnode*) to the first node of block *b'* (the variable *firstnode'*).

In function *create_vfg_nodes*, it traverses the instructions in block *b* and creates new appropriate nodes for different kinds of VFG-related instructions (Line 3-11). It also creates the edges between vfg nodes to represent the sequential order of the instructions (Line 15). There is a special situation in the process of constructing VFG nodes for each block. That is, a block does not contain any VFG-related instruction. In this case, ignoring the block (not creating a node) will cause the control flow information related to this block to be lost. To deal with that, we define a temporary structure called *NopNode* as the corresponding VFG node for this kind of blocks. At last, we traverse the NopNodes one by one and delete the NopNodes by connecting its predecessors and its successors, in order to *minimize* the *vfg*.

Example 4. Consider the method *CameraActivity.release*() in Example 1 again, whose CFG is shown in Fig. 8a. In the CFG, *release-B1* and *release-B7* indicate the entry and exit blocks. The block *release-B2* and *release-B3* represent the two branches of the first *if*-statement (Line 7-8 in Fig. 3), and the block *release-B5* and *release-B6* correspond to the second if-statement (Line 9-11). There are two resource release statements in the block *release-B5*: *mCamera.stopPreview*() and *mCamera.release*(). So in the VFG, block *release-B5* is divided into two FreeNodes (FreeNode1 and FreeNode2). We construct four NopNodes (NopNode1, NopNode2, NopNode3, and NopNode4) to indicate the blocks in the CFG that contain no VFG-related instructions. The original (before minimization) VFG is given in Fig. 8b. Then we delete all the NopNodes to minimize the VFG size. For NopNode1, we construct edges from its predecessor (RootNode) to its successor (NopNode3), and delete NopNode1. Similarly, we can delete NopNode2, NopNode3, and NopNode4. After minimization, we get the ultimate VFG as shown in Fig. 8c.

The CFG contains seven blocks and eight edges, while the VFG only contains four nodes and four edges. Furthermore, there are four paths in the CFG, and only two paths in the VFG. It can be observed that the complexity of a VFG is indeed smaller than that of a CFG.

create_vfg_nodes(*b*)

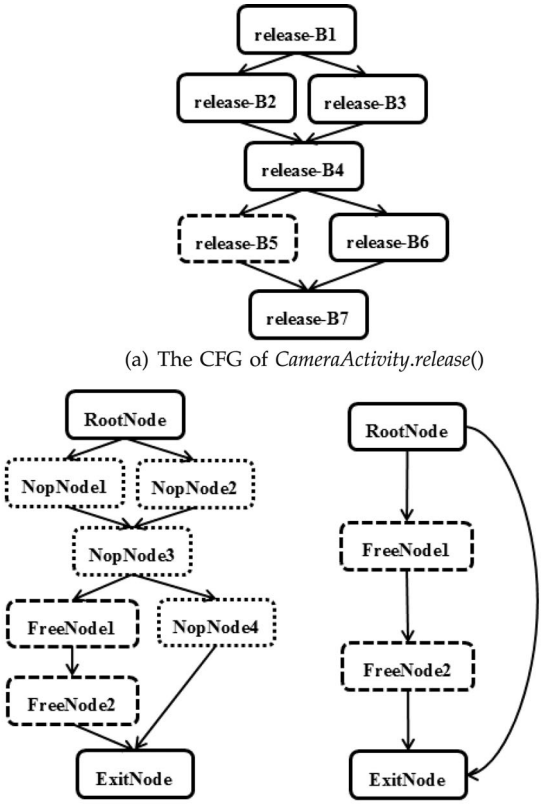
```
1: firstnode = lastnode = null
2: for each instruction ins in block b do
3:    if ins invokes a resource request API then
4:       create a SourceNode node
5:    else if ins invokes a resource release API then
6:       create a FreeNode node
7:    else if ins invokes a resource-irrelevant function then
8:       create a CallNode node
9:    else if ins is a "return" instruction then
10:      create an ExitNode node
11:   end if
12:   if firstnode is null then
13:      firstnode = lastnode = node
14:   else
15:      create an edge from lastnode to node
16:      lastnode = node
17:   end if
18: end for
19: if there is no VFG-related node in block b then
20:    create a NopNode node
21:    firstnode = lastnode = node
22: end if
23: return (firstnode, lastnode)
```

7.3 Model Checking

In this section, we explain how to use model checking technique to detect resource leaks in a VFG automatically. Now we assume there are m resources in our resource table. For the ith resource, we define propositions p_i and q_i to denote the invocation of the instructions for requesting and releasing the ith resource, respectively. The underlying set of atomic propositions AP is defined as follows:

$$AP = \{p_i \mid 1 \leq i \leq m\} \cup \{q_i \mid 1 \leq i \leq m\}.$$

Then let us consider the CTL formulas to describe the behaviour of resource leak. For a given LKS and the ith resource, if we claim that there is no resource leak about the ith resource, then the following property must hold: if there is a state s requesting the ith resource, then there must exist



(a) The CFG of *CameraActivity.release*()

(b) The original VFG of *CameraActivity.release*()

(c) The ultimate VFG of *CameraActivity.release*()

Fig. 8. The CFG and VFG of *CameraActivity.release*().

another state, which releases the ith resource, in every path from state s. It is obvious that CTL formula $AG(p_i \rightarrow AF(q_i))$ can describe this property. Therefore, the complete CTL formula set for all the resources is shown as follows:

$$\mathcal{F} = \{AG(p_i \rightarrow AF(q_i))|\ p_i \text{ occurs in the LKS}\}.$$

Given a VFG V, we can transform it to an LKS $M = \langle S, \Delta, I, L\rangle$ through the following steps. First, we generate the corresponding state and transition in M for each node and edge in V, respectively. Then we traverse the nodes in V one by one. Let n denote a node in the VFG, and its corresponding state in M is denoted by s. We have the following transformation rules.

- If n is a RootNode, we add s into I;
- If n is a resource related node (SourceNode or FreeNode), we add its corresponding propositions into $L[s]$ (atomic propositions that hold in state s);
- If node n is a CallNode, we add the resource summary of the called LKS into $L[s]$.

Finally, we apply a mature model checking tool to determine whether the formulas ($\mathcal{F}$) are satisfied in the LKS.

7.4 Multi-Threading Technique to Improve the Efficiency

Although using VFGs can decrease the complexity of analysis, the execution time is still a little long to analyze a relatively large app. Fortunately, we find that the execution time can be reduced further by distributing the analysis workload to multiple threads (processes) or even multiple machines. To implement this, we group the VFGs into several disjoint sets, where each VFG of one set does not call other VFGs in the same set directly or indirectly. As a result, the VFGs of the same set can be analyzed concurrently.

We use the topological ordering algorithm (Algorithm 4) to classify the VFGs. It first initializes a list *vfg_sets* as an empty list, that is used to store the target disjoint VFG sets. Then it enters a while-loop. In each iteration, it collects the VFGs whose outdegrees are zero, into a set (the variable *candidate*). It removes the VFGs in *candidate* from the VFG list (the variable *vfg_list*) and updates the outdegrees of the parents of these VFGs (Line 5-8). Then, it adds *candidate* into *vfg_sets*, and starts new iterations until *vfg_list* is empty. The list *vfg_sets* organizes the sets in such a way that the VFGs in the former set should be checked before that in the latter set.

Algorithm 4. Group the VFGs

```
1: vfg_sets = []
2: while vfg_list is not empty do
3:    candidate = ∅
4:    add the VFGs with zero outdegrees into candidate
5:    for each vfg in candidate do
6:       remove vfg from vfg_list
7:       decrease outdegrees of vfg's parents by 1
8:    end for
9:    append candidate to vfg_sets
10: end while
```

Recall the FCG in Fig. 6. We can divide the methods into three sets as follows. We should first analyze the method set S_1 and then S_2 and S_3 in order.

$$\begin{aligned} S_1 &= \{\textit{Camera.open}, \textit{Camera.startPreview}, \\ &\quad \textit{Camera.stopPreview}, \textit{Camera.stop}, \\ &\quad \textit{Camera.startFaceDetection}, \\ &\quad \textit{MainActivity.onCreate}\} \\ S_2 &= \{\textit{CameraActivity.takePicture}, \\ &\quad \textit{CameraActivity.release}, \\ &\quad \textit{CameraActivity.onStart}\} \\ S_3 &= \{\textit{CameraActivity.onCreate}, \\ &\quad \textit{CameraActivity.onPause}\} \end{aligned}$$

8 Implementation and Evaluation

To evaluate the usefulness of our method, we developed an automated static analyzer called Relda2. All code in Relda2 is written in Python language, so it is easy to be deployed on different operating systems, such as Linux and Windows. Specifically, Relda2 is implemented on top of `Androguard` that maps DEX/APK format into full Python objects. In addition, `Androguard` provides the basic static analysis of the code (blocks, instructions, and permissions). The model checker used here is NuSMV [38], which is a symbolic model checker for CTL. Based on these tools, we implemented the resource leak detection techniques in Relda2.

Compared with our previous tool Relda [1], we implemented three main techniques in Relda2: (1) construct a more precise FCG with the CBG, (2) implement the flow-sensitive detection technique to eliminate a number of false negatives, and (3) use the multi-threading technique to accelerate the analysis efficiency. In order to evaluate the effectiveness of these key techniques, we need to answer the following research questions:

TABLE 3
Different Combinations of Techniques in Relda2

options	CBG	flow-sensitive	multi-threading
op_1	✘	✘	✘
op_2	✓	✘	✘
op_3	✓	✓	✘
op_4	✓	✓	✓

- **RQ1:** Is the FCG, constructed with the CBG, useful to eliminate the useless methods in the app?
- **RQ2:** In the flow-sensitive detection technique, is the VFG's size smaller than that of the CFG? What is the percentage of decrease in the size?
- **RQ3:** Compared with the flow-insensitive analysis, can the flow-sensitive technique find more resource leaks?
- **RQ4:** Can the multi-threading technique accelerate the efficiency of the analysis?

To answer the research questions, we implemented the key techniques in Relda2 as several options. That is, we can configure Relda2 to use different technique combinations. In Table 3, "✘" indicates that Relda2 does not use the technique. In contrast, "✓" means Relda2 employs the technique. In fact, op_1 is the same as Relda.

In the following, we first describe our experimental subjects, and then discuss our empirical results to answer these research questions. Afterwards, we will make a summary of resource leaks in real Android apps, which mainly includes the statistics of the most common resources unreleased in apps and the major causes of resource leaks.

8.1 Experimental Applications and Setup

Totally, we collected 103 real apps for our evaluation, where 90 apps are from two famous official app markets, Google Play [23], and Wandoujia [24] (a popular Android app market in China), and 13 apps are from an open-source Android app repository [25]. For the apps from app markets (closed-source), we only downloaded the installer files (.apk files), while we downloaded the install files and source code of the apps from the open source repository. The functionality of these apps are varied (like sociality, media, game), but they all use several exotic resources.

Relda2 reported that 76 apps (68 closed-source apps and 8 open-source apps) might have resource leaks, and we tried to confirm these bug reports. However, there are currently no specific tools publicly available to confirm whether the resource leaks in our bug reports are real bugs. Here, we mainly used error-guessing testing to design appropriate test cases that can exactly trigger the resource leaks reported by Relda2 (note that this technique can just help us confirm a part of the reports, the rest ones may also be real resource leaks). Specifically, for the open source apps, we manually review the source code to find whether the apps miss the resource release operations. Based on our understanding of the source code, we carefully selected the test cases that can trigger the bugs. With the help of source codes, we can easily figure out which bug reports are real in several minutes. However, for the closed-source apps, we can only design appropriate test cases according to the possible locations of the bugs provided by Relda2.

TABLE 4
The Statistics of Experimental Applications

origin	*app*	*size*	*dex*	*#class*	*#method*
open-source community	Andless	0.3	0.1	69	399
	Bluechat	0.1	0.04	47	154
	Foocam	0.3	0.6	507	3347
	Getbackgps	0.3	0.2	69	442
	Impeller	2.8	4.5	3858	28178
	Runnerup	1.9	4.4	1107	5666
	Visualizer	1.2	0.02	23	76
	Yaacc	1.6	3.0	2379	16418
app markets	Anydo	10.8	6.6	6926	45985
	Apollo	2.4	2.8	1571	12169
	Applock	0.4	0.4	428	2421
	ATorch	0.7	0.4	180	1132
	BaibianTorch	2.6	1.5	736	4747
	BeautyCamera	16	3.7	4094	24210
	BluetoothHelper	1.6	1.0	781	4989
	BubeiListen	3.8	2.5	2804	15966
	BudingTicket	3.8	2.2	1948	14727
	Checkcheck	4.5	1.5	1002	6343
	CheckLove	2.1	1.5	773	4345
	Compass	1.4	0.8	472	3449
	Guodegang	2.1	1.8	1516	9562
	HikiPlayer	0.2	0.2	164	1062
	HuaxiaZitingShu	4.5	0.8	875	4569
	Ibody	5.2	2.5	2702	15768
	Instagram	9.5	4.0	4725	28511
	Itake	0.8	0.3	349	2470
	Jane	42.7	5.1	4572	31631
	KuaipaiQR	5.5	4.8	3385	20113
	Linijka	0.2	0.2	200	1138
	MiguMusic	13	7.5	6787	39703
	Mima	2.3	3.4	3253	21438
	PicsArt	18.3	7.4	7052	43838
	QRScan	1.6	0.5	429	2378
	Shopsavvy	5.3	1.9	1563	9245
	SimSimi	3.9	4.8	5398	33401
	TianyiVideo	4.7	4.7	4905	29845
	TigerMap	3.7	1.7	1736	9203
	TimeCounter	0.2	0.1	47	313
	UTorch	1.0	0.6	787	3863
	VPlayer	6.6	2.4	2046	14973
	WeatherPro	5.0	0.9	491	2964
	Whospy	0.7	0.5	208	924
	Yelp	14.3	5.2	6024	35809

As the resource leaks can not be easily observed from runtime information, we developed an auxiliary tool called *AddStake* to record the resource-related operations when executing a test case. This tool is based on program instrumentation, that inserts a number of extra instructions into the app to record the needed information. The workflow of *AddStake* is like this: it first unpacks the app to get the smali files, and then inserts our monitoring instructions into these files. Finally, it repacks the smali files back to apk files. Because of some protection mechanisms, it will fail sometimes in the repacking step (e.g., the two apps in Section 8.8). Among 68 closed-source apps, the tool can successfully deal with 35 apps. Finally, we get 43 apps (35 closed-source and 8 open-source apps) as experimental apps, whose information is shown in Table 4. In this table, *app* in the first column denotes the names of the apps, *size* is the size of the apk file (MB), *dex size* is the size of the dex file (MB). The

```
public class ClientActivity extends Activity {
   private BluetoothAdapter mBluetoothAdapter;
   protected void onCreate(Bundle
                  savedInstanceState) {
      ......
      mBluetoothAdapter =
         BluetoothAdapter.getDefaultAdapter();
      ......
      startDeviceSearch();
   }
   private void startDeviceSearch() {
      mBluetoothAdapter.enable();
      ......
   }
}
```

Fig. 9. Bluechat example.

sizes of our experimental apps range from tens of KB to dozens of MB. The last two columns show the number of classes and methods in an app. In this section, all experiments are performed on an Intel Xeon 2.40 GHz (16 cores) machine, with 32 GB memory and CentOS 6.5 operating system.

8.2 A Case Study

In this section, we present a case study. *Bluechat* is an open-source and light-weight P2P chat app that uses the energy-consuming resource *Bluetooth*. Two devices connect each other with bluetooth and one can send messages to another. Fig. 9 shows a simplified code snippet of the process.

As soon as *ClientActivity* is activated, it first obtains an instance of BluetoothAdapter (the variable *mBluetoothAdapter*) in the method *onCreate*(), which is used to perform fundamental Bluetooth tasks. Then it invokes the method *startDeviceSearch*() to search for the nearby devices. At the beginning of the method *startDeviceSearch*(), it turns on the local Bluetooth adaptor by calling method *mBluetoothAdapter.enable*(). However, the developers forgot to turn off the Bluetooth adaptor in this Activity.

In our resource summary process, we first get the summary of the method *startDeviceSearch* as [*BluetoothAdapter.enable*]. By the inter-procedural analysis, we get the summary of the method *onCreate* as [*BluetoothAdapter.enable*]. After that, we consider implicit order of callbacks (mainly about lifecycle), and we found there is no method in the app, whose resource summary contains *BluetoothAdapter.disable* to turn off the local Bluetooth adapter. Note that both of our analysis approaches (flow-insensitive and flow-sensitive) are able to detect this resource leak.

8.3 RQ1: Can FCG Eliminate the Useless Methods?

To answer this question, we carry out the following steps:

- First, we configure Relda2 with op_1 and op_2 to construct FCGs for each app. The FCGs generated by Relda2 with op_1 and op_2 are fcg_1 and fcg_2, respectively.
- Then we compare the functions in fcg_1 and fcg_2 and determine whether some functions in them are useless through manual review of the code.

To determine whether some functions in the FCG generated by Relda are useless, we manually review the source code of several experimental apps. For the closed-source app, we first use a reverse engineering tool called dex2jar [19] to decompile an app (.apk) into a jar file (.class). Then we employ a standalone graphical utility JD-GUI [39] to display Java source code of each class. Finally, we find that there are a number of useless functions in fcg_1. We classify the main causes of those useless functions as the following categories:

- There are a number of official packages to make apps downward compatible. Their names always start with "android.support". In practice, it often happens that a lot of programmers tend to include them into their apps, even though the apps do not use any class of these official packages.
- Some apps include third-party packages. For instance, it is a trend that most apps are embedded with social contact functionality. It is a convenient way to implement this functionality by importing some mature SDKs that are provided by famous social contact services, such as Facebook, Twitter, or WeChat in China. For some reasons, the apps may not actually use these third-party packages.
- With the app upgrade, some parts of code in the new version of the app may be not executed anymore, that is, they become "dead code". The developers may not delete them from the new apps in time.

8.4 RQ2: Is VFG's Size Smaller than that of CFG?

For RQ2, we first try to measure the number of nodes (blocks) and edges in VFGs and CFGs for each app. We observe that the numbers of nodes in VFGs are just a little less than that of CFGs in most apps. In some special apps, the number of nodes in VFG even exceeds that in CFG. Recall the example in Fig. 8, a CFG node will be separated into multiple nodes in VFG when the CFG node has multiple VFG-related statements. However, according to the experimental results, we also observe that the analysis efficiency depends on the number of paths or branches in the analysis model, rather than the number of nodes or edges of the model. Therefore, we use the Cyclomatic Complexity (CC) developed by McCabe [40] to measure the complexity of a program. Here, the CC of an app is calculated by aggregating that of each method.

Table 5 shows the details of our experimental results. The first column gives the app names, and the next two columns show the number of instructions in CFGs and VFGs for each app. The fourth column gives the reduced percentage of instructions in VFGs. The following two columns show the CC of CFGs and VFGs for each app. The last column gives the reduced percentage of the CC in VFGs. From Table 5, it can be observed that the percentage of decrease in the number of instructions ranges from 75.3 to 90.5 percent (84.3 percent on average). The percentage of the CC decreases ranges from 22.9 to 54.9 percent (38.3 percent on average). These observations show that VFG can effectively decrease the complexity of the analysis model.

8.5 RQ3: Can the Flow-Sensitive Technique Find More Real Resource Leaks?

For RQ3, we configure Relda2 with options op_2 and op_3 to analyze apps and get bug reports, respectively. Table 6 shows the detection results of the experimental apps. In this

TABLE 5
The Effectiveness of VFG

app	*#CFG_INS*	*#VFG_INS*	*ins_reduced(%)*	*#CFG_CC*	*#VFG_CC*	*cc_reduced(%)*
Andless	8469	781	90.1	765	542	29.2
Bluechat	1585	184	88.4	182	123	32.4
Foocam	1360	129	90.5	116	73	37.1
Getbackgps	2968	488	83.6	446	201	54.9
Impeller	5846	800	86.3	766	496	35.2
Runnerup	24982	2925	88.3	2498	1294	48.2
Visualizer	233	32	86.3	28	13	53.6
Yaacc	13133	1367	89.6	1358	702	48.3
Anydo	40762	8153	80.0	4377	2963	32.3
Apollo	19039	4216	77.9	2472	1472	40.4
Applock	8921	1235	86.2	779	501	35.7
ATorch	8921	1235	86.1	779	501	35.7
BaibianTorch	24443	3997	83.6	2154	1661	22.9
BeautyCamera	102819	12768	87.6	11234	6162	45.1
BluetoothHelper	46098	6950	84.9	4558	3234	29.0
BubeiListen	51322	9365	81.8	5672	3715	34.5
BudingTicket	42295	7594	82.0	4249	2310	45.6
Checkcheck	37844	6387	83.1	3921	2495	36.4
CheckLove	18304	3478	81.0	1743	941	46.0
Compass	8340	1116	86.6	895	519	42.0
Guodegang	31278	3395	89.1	2801	1482	47.1
HikiPlayer	8747	1729	80.0	1148	877	23.6
HuaxiaZitingshu	12539	1498	88.1	1315	696	47.1
Ibody	59749	7924	86.7	6316	3773	42.3
Instagram	16854	4169	75.3	2411	1532	36.5
Itake	6076	994	83.6	633	424	33.0
Jane	32655	4881	85.1	3702	2372	35.9
KuaipaiQR	73710	7644	89.6	6866	3777	45.0
Linijka	4384	654	85.1	500	298	40.4
MiguMusic	106795	15673	85.3	12790	8056	37.0
Mima	33287	5133	84.6	3706	2404	34.7
PicsArt	194930	38394	80.3	21453	14380	33.0
QRScan	14922	2314	84.5	1620	1098	32.2
Shopsavvy	15454	2946	80.9	1968	1259	36.0
SimSimi	151665	26373	82.6	16183	10429	35.6
TianyiVideo	93012	19693	78.8	10343	6680	35.4
TigerMap	53030	9545	82.0	5663	4096	27.7
TimeCounter	3081	563	81.7	421	218	48.2
UTorch	5443	724	86.7	555	302	45.6
VPlayer	15178	2886	81.0	1684	1133	32.7
WeatherPro	24794	4006	83.8	2543	1773	30.3
Whospy	4837	968	80.0	520	335	35.6
Yelp	54536	12144	77.7	6859	4324	37.0

table, *app* in the first column denotes the names of the apps. The next five columns and the last six columns show the detailed results generated by Relda2 with the option op_2 and the option op_3.

- pt indicates the preprocessing time, which includes the time of FCG construction and CFG construction.
- vt means the time of VFG construction.
- st denotes the time spend in the resource summary process.
- it is the implicit callback handling time.
- #TP is the number of true positives we confirm with error-guessing testing.
- #REP is the total number of leaks reported by Relda2.

Here, the way we count the values of #TP and #REP is a little different from that we used in our previous work, which treats each path containing unreleased resources as an independent resource leak. Through further investigation, we observe that many of the leaks in different paths are all caused by the same resource request statements in the app, that can be fixed uniformly by inserting release operations in the same exit points. Therefore, we merge these redundant reports into one in Relda2. We make two observations in the following sections.

8.5.1 True Positives

From Table 6, we can see that Relda2 is able to detect real resource leaks in these Android apps. Totally, Relda2 with option op_2 (flow-insensitive) detects 69 potential resource leaks with 47 true positives, while Relda2 with option op_3 (flow-sensitive) detects 121 potential resource leaks with 67 true positives. The precision of the flow-insensitive technique reaches 47/69 = 68.1%, and that of the flow-sensitive technique is 67/121 = 55.4%. Since our error-guessing testing is incomplete, the real precision rate could be greater

TABLE 6
Results of the Flow-Insensitive and Flow-Sensitive Approach

App	Relda2 with op_2					Relda2 with op_3					
	pt (s)	*st* (s)	*it* (s)	*mem* (MB)	#TP/#REP	*pt* (s)	*vt* (s)	*st* (s)	*it* (s)	*mem* (MB)	#TP/#REP
Andless	0.3	0.01	0.03	24	1/2	2.3	0.1	5.8	0.03	37	1/3
Bluechat	0.09	0.001	0.007	19	1/1	0.2	0.02	2.3	0.01	22	1/1
Foocam	0.68	0	0.001	59	1/1	0.77	0.02	1.5	0.002	61	1/2
Getbackgps	0.16	0.002	0.02	24	1/3	0.4	0.05	3.9	0.02	30	1/3
Impeller	10.6	0.004	0.13	381	2/2	11	0.09	22.2	0.13	393	2/2
Runnerup	2.1	0.01	0.6	105	0/1	4.5	0.35	34.8	0.63	145	0/3
Visualizer	0.03	0	0	18	1/1	0.05	0.003	0.37	0	18	1/2
Yaacc	5.4	0.01	0.19	230	1/1	6.69	0.2	28.2	0.19	253	1/2
Anydo	25	0.01	0.09	583	1/1	29	0.5	126	0.09	650	1/1
Apollo	3.6	0.02	0.6	185	2/3	5.9	0.3	38	0.6	222	2/4
Applock	0.7	0.01	0.01	48	1/1	1.6	0.1	7.1	0.01	61	1/1
ATorch	0.3	0.003	0.04	35	2/2	0.7	0.06	5.2	0.05	43	2/2
BaibianTorch	1.5	0.01	0.01	81	4/4	4.1	0.3	20.7	0.1	118	5/5
BeautyCamera	11.9	0.06	6.8	339	0/0	28.0	1.6	258.5	6.6	519	0/1
BluetoothHelper	2.0	0.02	0.54	81	0/1	14.4	0.64	48.6	0.56	160	0/1
BubeiListen	6.1	0.01	1.87	211	3/4	12.6	0.76	91	1.9	300	4/5
BudingTicket	4.3	0.01	1	185	0/1	8.9	0.5	67	1	247	1/2
Checkcheck	2.3	0.02	0.97	111	3/3	7.3	0.5	50	1	173	4/6
CheckLove	1.4	0.01	0.5	77	1/1	4.2	0.4	27	0.5	108	1/1
Compass	0.87	0.005	0.07	62	0/0	2.2	0.12	10.3	0.07	77	1/1
Guodegang	3.2	0.02	0.68	147	1/1	6.2	0.46	45.8	0.69	198	2/2
HikiPlayer	0.33	0.01	0.01	30	0/0	1.4	0.1	7.2	0.01	45	2/4
HuaxiaZitingshu	1.3	0.007	0.10	75	1/1	2.5	0.18	15.1	0.1	96	2/2
Ibody	6.2	0.03	2.06	212	1/2	13.6	0.87	98.8	2.06	315	2/4
Instagram	12.2	0.01	0.21	355	1/1	14.5	0.26	61.3	0.22	391	1/3
Itake	0.4	0.01	0.02	35	3/3	1.2	0.08	6.1	0.03	45	3/5
Jane	12.6	0.02	0.44	395	0/0	18.8	0.45	65.4	0.44	450	0/1
KuaipaiQR	8.8	0.04	4.2	283	2/4	16.9	1.1	140	4.2	400	3/6
Linijka	0.3	0.002	0.01	31	1/1	0.7	0.06	5.3	0.02	39	1/1
MiguMusic	21.6	0.06	6.4	546	1/3	39.1	1.52	266.6	6.38	732	1/3
Mima	7.7	0.02	0.74	279	2/3	12.22	0.46	61.8	0.73	339	4/6
PicsArt	28.7	0.12	28.5	598	2/2	57.9	3	728.3	28	935	2/2
QRScan	0.8	0.006	0.1	50	0/4	2.6	0.2	14.1	0.1	76	0/6
Shopsavvy	2.86	0.01	0.33	152	2/2	4.3	0.21	24	0.34	177	2/4
SimSimi	15.4	0.03	2.3	369	0/1	28.4	1.0	141	2.3	483	0/1
TianyiVideo	14.35	0.01	7.57	406	1/3	26.1	1.8	284.5	7.6	564	1/3
TigerMap	3.56	0.01	0.76	143	1/1	13.77	0.72	67.12	0.76	228	2/6
TimerCounter	0.13	0.001	0.003	22	0/0	2.88	0.04	2.82	0.006	27	2/2
UTorch	1.0	0.003	0.03	64	0/0	1.6	0.1	5.1	0.3	73	2/3
Vplayer	4.58	0.007	0.16	196	1/1	6.64	0.21	23.81	0.17	222	1/2
WeatherPro	1.52	0.01	0.32	91	0/1	6.18	0.43	25.34	0.33	131	1/2
Whospy	0.29	0.002	0.01	30	1/1	1.08	0.07	5.01	0.02	39	1/1
Yelp	19.01	0.04	3.4	429	1/1	23.64	0.79	181.8	3.38	527	2/4

than that shown in the table. It is a fact that the set of leaks detected by the flow-insensitive technique is a subset of those detected by the flow-sensitive technique.

We have also reviewed the source code of several experimental apps, and found some resource leaks reported by the flow-sensitive approach are indeed false positives. Through further analysis, we found a major reason is that Relda2 does not check the data dependency precisely, due to the concern of the cost of analysis. We show two of the scenarios, where the operations on resources are related to the runtime state of some decision variables.

- If a resource is not requested successfully because of some exception, then it is not necessary to release it.
- Developers may use some flag variables to determine whether the resources are requested.

8.5.2 Execution Time

Now we discuss the efficiency of our approach. Consider the second and seventh column in Table 6 that show the preprocessing time of Relda2 with option op_2 and op_3, respectively. The preprocessing time of flow-insensitive analysis is less than that of flow-sensitive analysis due to the absence of CFG construction process. The eighth column shows that the overhead of VFG construction is rather low. The third and ninth column show the analysis time of the two techniques. The execution time of flow-insensitive analysis is within one minute for each app, and it costs 6.7 seconds on average. The flow-sensitive technique costs at most 817 seconds (about 14 minutes), and 79 seconds (about 1.3 minutes) on average. The analysis time of our two detection techniques are both acceptable to the testers.

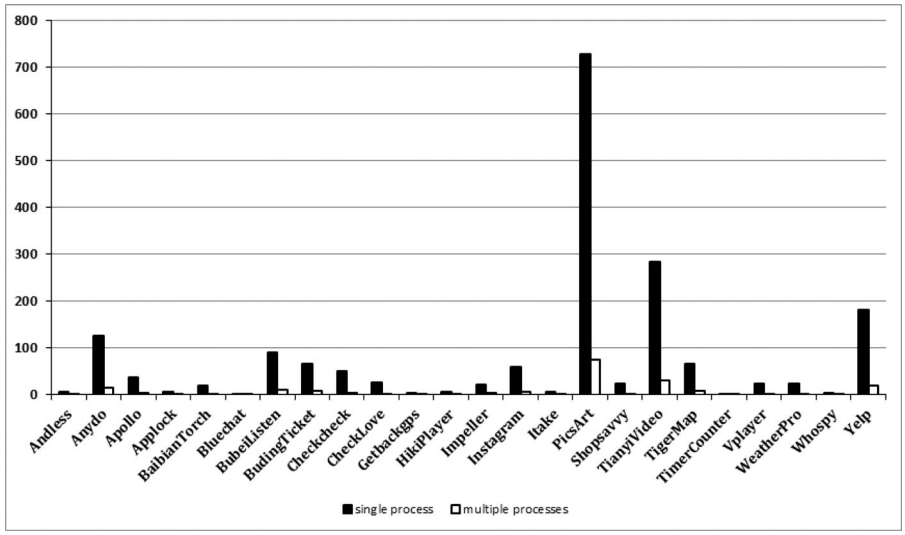


Fig. 10. Resource summary time comparison between single and multiple processes.

8.6 RQ4: Can the Multi-Threading Technique Accelerate the Efficiency of the Analysis?

From Table 6, we see that the maximum resource summary time of the flow-sensitive analysis reaches 728 seconds (about 12 minutes, app *PicsArt*), so we propose a multi-thread algorithm to accelerate the analysis efficiency. To verify its effectiveness, we configure Relda2 with options op_3 and op_4 to analyze the experimental apps. We use the module *multiprocessing* in Python to implement our multi-threading technique, and set the number of processes from 1 to 16 (our machine has 16 CPU cores).

Fig. 11 shows the resource summary time under different number of processes. From this figure, we can easily observe that the time is significantly reduced by the multi-threading technique, and we give the empirical value for the number of processes as 8, because there is little improvement (in execution time) when the number exceeds 8. Fig. 10 compares the time of single and multiple processes (eight processes). The resource summary time of the single thread method is often five to seven times as much as that of the multiple thread method. The maximum time of the resource summary process decreases to less than 1.5 minutes. These facts show that multi-thread technique does enhance the efficiency of the flow-sensitive analysis significantly.

8.7 Comparison with Dynamic Execution

Besides static analysis, dynamic execution with logging is an intuitive approach to detect the resource leaks. In this section, we apply Monkey [41], a testing tool provided by Android system, to automatically generate and execute test cases. We also employ our tool *AddStake* for instrumentation to trace the information of resource operations at runtime. We set the number of events for Monkey as 10,000. Finally, we review the log files containing the resource operations to find the potential bugs.

In fact, it is hard to make a strict comparison between the dynamic and static approaches. The current dynamic analysis techniques for Android are not effective enough for covering the elements of programs. Therefore, we give a rough comparison in Table 7, where #E denotes the number of events successfully executed for each app; #T shows the execution time; #D represents the number of resource leaks detected by this approach; and #C is the number of resource leaks that are reported by Relda2 and have been confirmed (See the last column in Table 6). The values of #E are not always equal to 10,000, because Monkey may fail to execute some events and then it exits. Comparing the last two columns, we can observe that the dynamic execution technique only detects about one fifth (13/67 = 19.4%) real resource leaks in the experimental apps. In addition, the resource leaks detected by Monkey are all reported by our tool Relda2. Through further analysis, we get one of the possible reasons is that it is difficult and time-consuming to automatically generate test cases that can execute the specific program paths containing resource leaks.

8.8 Experiments in Industry

In addition to the apps from app markets and open source communities, we have also contacted the developers in several famous companies in China. We have analyzed some apps about their core business. In these apps, our tool detected a number of resource leaks, a part of which have been confirmed by their developers. Some of them affect user experience badly. For instance, we have found a leak of *LocationManager* in a map app that will drain users' battery energy in a few hours. Another resource leak about *PowerManager.WakeLock* was found in an e-commerce app, that keeps the phone's screen always on. We do not give the detailed information about these leaks in this paper.

8.9 The Summary of Resource Leaks in Real Apps

In this section, we will make a summary of resource leaks in real Android apps, that mainly includes the statistics of the most common resources unreleased in apps, and the cause of resource leaks. It will let developers know which resources are commonly used incorrectly, and they should be careful when using these resources.

8.9.1 Most Common Unreleased Resources in Apps

Fig. 12 shows the statistics of the most common unreleased resources in apps. We can observe that 30 percent of such

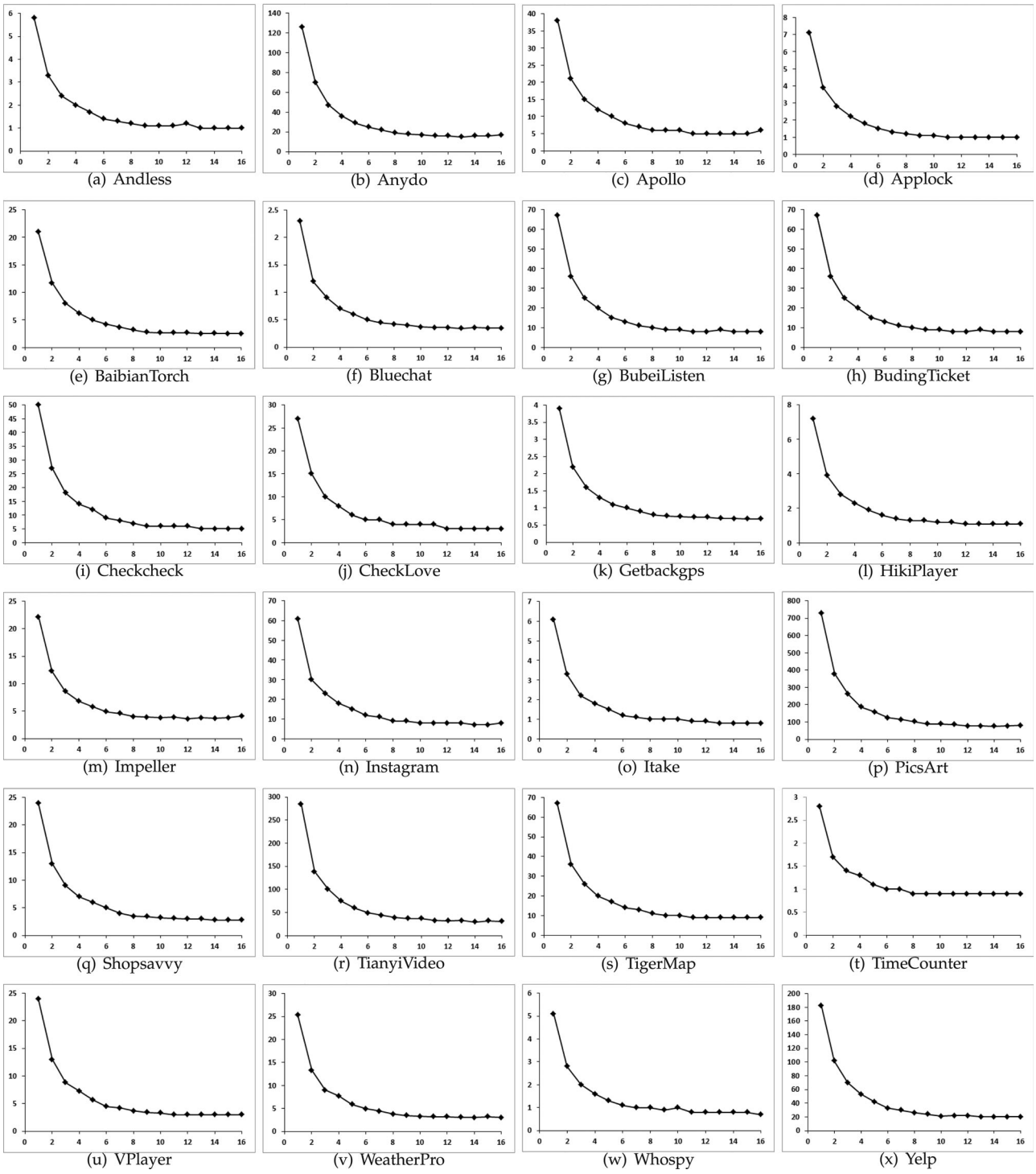


Fig. 11. Resource summary time under different number of processes.

leaks are related to MediaPlayer, 30 percent of them are related to Camera, and 13 percent of them are related to PowerManager. Table 8 shows the corresponding resource requesting APIs.

8.9.2 The Cause of Resource Leaks

To figure out the cause of resource leaks, we also manually reviewed the source code of several experimental apps. We classify the causes for resource leaks as the following categories:

- *Releasing resources in user-triggered callbacks which may not be invoked*. This is the major reason of resource leaks. In this case, developers try to release resources in user triggered callbacks, which are not necessarily invoked before the app finishes. A good suggestion is to release them in their releasing callbacks, as we pointed out in Table 2.
- *Forgetting to release a resource*. The cause of these leaks is that the programmer simply forgets to release a requested resource throughout the code. Although it

TABLE 7
Results of Monkey

App	#E	#T (s)	#D	#C
Andless	10000	139	0	1
Bluechat	10000	121	0	1
Foocam	10000	156	0	1
Getbackgps	10000	210	0	1
Impeller	10000	183	0	2
Runnerup	10000	169	0	0
Visualizer	6750	254	0	1
Yaacc	10000	112	1	1
Anydo	10000	220	0	1
Apollo	2901	109	1	2
Applock	10000	228	0	1
ATorch	10000	189	0	2
BaibianTorch	10000	88	3	5
BeautyCamera	10000	148	0	0
BluetoothHelper	10000	132	0	0
BubeiListen	3944	134	2	4
BudingTicket	10000	344	0	1
Checkcheck	10000	138	0	4
CheckLove	10000	245	0	1
Compass	10000	176	0	1
GuoDeGang	10000	212	0	2
HikiPlayer	10000	215	0	2
HuaxiaZitingshu	10000	110	0	2
Ibody	10000	271	0	2
Instagram	10000	172	0	1
Itake	10000	318	1	3
Jane	10000	313	0	0
KuaipaiQR	7415	66	1	3
Linijka	10000	108	0	1
MiguMusic	5227	124	0	1
Mima	10000	255	1	4
PicsArt	10000	293	0	2
QRScan	10000	213	0	0
Shopsavvy	10000	256	0	2
SimSimi	2564	55	0	0
TianyiVideo	10000	167	0	1
TigerMap	10000	176	0	2
TimeCounter	10000	209	1	2
UTorch	10000	186	0	2
VPlayer	10000	190	1	1
WeatherPro	10000	235	1	1
Whospy	10000	152	0	1
Yelp	1856	64	0	2

seems like a simple mistake, this does happen in real apps.

- *Failing to release a resource in callbacks of all the implemented interfaces.* Due to Android's event-driven nature and the large number of user callbacks used by Android framework, the execution paths of Android apps are often complicated. Even a careful programmer can easily fail to release all resources along all possible invocations of events or callbacks. For a resource requested in an ordinary class, it must be released in at least one callback of all interfaces implemented by the classes.
- *Failing to understand the lifecycle of Android apps.* In this case, developers release resources only when the app finally exits, that is, they only release them in *onDestroy*() of the activity. *OnDestroy*() is only called when the app component is about to be destroyed. However, when the user exits any app (press BACK button), Android framework still maintains the state of the app to reduce the restart time of the app, instead of destroying the app with the *onDestroy* callback. It essentially means that the app may not actually be destroyed; it may hold the resources for a long time.

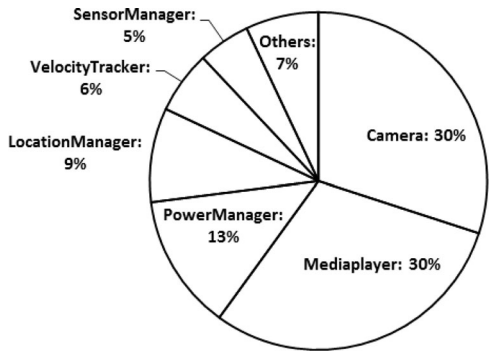


Fig. 12. Statistics of the most common unreleased resources in apps.

9 Related Works

Our study in this paper is related to several research topics, including resource leak detection, testing and program analysis for Java programs and Android apps, and typestate related analysis. In this section, we will discuss some of these works in recent years.

9.1 Resource Leak Detection

Many researchers focus on detecting memory leaks in managed code [42], such as Java and C#. Mitchell and Sevitsky [43] presented a tool called Leakbot to detect memory leaks in Java programs by formulating structural and temporal properties of reference graphs. Xu et al. [44], [45], [46] also focused on memory leaks in Java programs. They proposed several approaches based on some observation (about containers, loops) to optimize the static analysis to produce precise bug reports at an acceptable cost. Bond and Mckinley [47] proposed an approach called leak pruning, which predicts dead objects and reclaims them based on observing data structure usage patterns. The memory leak problem is similar to resource leaks. Both of them focus on several pairs of specific APIs, such as new/delete APIs for memory leaks, and request/release APIs for resource leaks. However, there are several differences between the detection approaches for these two problems. The major issue of detecting memory leaks is to monitor or simulate the usage of memory. To perform accurate analysis, it often needs to construct a memory model that can correctly

TABLE 8
Summary of the Most Common Unreleased Resources in Apps

Unreleased Resource	Request Method
MediaPlayer	new/create
MediaPlayer	start
Camera	open
Camera	startPreview
PowerManager.WakeLock	acquire

TABLE 9
Comparison between Our Work and Related Works

	Target object	*Aimed resources*	*Main techniques*
M. Arnold	Java byte-code	SWT resources and IO streams	runtime verification
E. Torlak	Java source code	system resources (such as sockets and database connections)	static analysis
N. Mitchell	Java source code	memory	static analysis (with reference graph)
G. Xu	Java source code	memory	dynamic analysis (with container profiling)
Y. Liu	Java byte-code translated from Android byte-code	sensor-related resources	runtime verification (with JPF)
A. Pathak	Java byte-code translated from Android byte-code	non-sleep related resources (WakeLock)	static analysis (with reaching definition data flow)
K. Kim	Android byte-code	non-sleep related resources (WakeLock)	runtime verification
Relda2	Android byte-code	general resources	static analysis and model checking

record whether each memory unit is used, when it comes to "new" or "delete" statements. In addition, this problem is related to the memory management mechanisms in different programming languages (like C, C++ and Java). Different with the memory leaks, the resource leak problem just considers the matches of resource-related operations instead of constructing memory models. Therefore, we can design a more lightweight and effective algorithm to detect it.

Arnold et al. [48], [49] designed a specialized runtime environment for detecting resource (such as SWT resource and IO stream) leaks in Java programs. The main idea of their work is to monitor the execution of the application and check for violations of resource safety properties. Torlak and and Chandra [50] implemented a tool called Tracker to perform inter-procedural static analysis for finding resource leaks in Java programs, and ensure that no resource safety policy is violated on any execution path. Weimer and Necula [51] presented a static data-flow analysis method for finding defects that the programs may not properly release the important resources in the presence of exceptional situations. These works only consider a few Java-platform-related resources, not Android-platform-related resources.

Liu et al. [52], [53] built a tool called GreenDroid on top of Java Path Finder (JPF) [54] to find sensor-related energy inefficiency problems in Android apps. Specifically, they simulate the runtime behavior of an app and analyze its sensory data utilization. Since they need to execute an Android app in JPF's Java virtual machine (JVM), their method can not deal with apk files directly.

Pathak et al. [3], [17] presented a study of energy bug characteristics. One of their conclusion indicates that a major cause of energy bugs is inappropriate request and release operations of resource *WakeLock*. In their works, they treated the acquiring and release of a WakeLock as a definition of assignment to a corresponding variable, thus detecting non-sleep code paths was converted to the reaching definition (RD) dataflow problem [55]. Kim and Cha [56] also focus on no-sleep energy bugs, but they detect this problem in runtime. They first analyze the WakeLock request and release mechanism in Android platform and then monitor the kernel functions to detect the WakeLock behavior.

Table 9 shows the details of comparison between our work and other resource leak related works. For Android apps, most works related to resource leak aim to find energy inefficiency problems [3], [17], [52], [53], [56], which have received much attention for the past several years. Similar to these 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. The major differences between our work and these works are two-fold. First, we systematically define the resource leak problem and collect the resource table. As discussed throughout this paper, the resources we focus on include not only energy-related resources, but also memory-related and exclusive resources. Second, we detect the resource leaks from Dalvik byte-code rather than source code or Java byte-code, since this way is more practical when the source code is not available to testers. Besides, our approach is based on classic static analysis and model checking, not runtime verification.

Currently, we collect the resources through summarizing the Android Reference manually. A possible improvement is that we can use Nature Language Processing (NLP) techniques [57] to automate the process.

9.2 Testing of Android Apps

Some existing studies have focused on testing Android applications to find potential bugs [7], [9], [10], [11], [12].

Fuzzing is one state-of-the-art approach for testing Android apps, and there is a popular fuzzing tool called Monkey, which is included in Android platform. Monkey can efficiently generate a lot of simple inputs, so Hu and Neamtiu [7] used it to find GUI bugs in Android apps. However, Monkey is not suitable for generating highly specific inputs, which are often required to cause resource leaks.

Yan et al. [10] proposed a model-based approach to generate test cases for detecting resource leaks in Android apps. There are various reverse engineering techniques ([58], [59], [60]) that can be used to construct GUI models automatically. Yan et al. used a tool called AndroidRipper [61], [62] to construct GUI models for Android applications dynamically. However, they found the models produced by AndroidRipper are very detailed (each GUI object may be included multiple times) so that the size of models is too large. To reduce model size, they created the models manually after examining the output of AndroidRipper and the

source code of the application. Note that the resources they focused on are memory, thread and binder.

Prior works have applied concolic execution to generate feasible sequences of events for Android apps [11], [12]. However, this approach suffers from the path explosion problem. In Anand's work [11], the maximum length of event sequences is 4. Mirzaei et al. [12] use static analysis to reduce the set of event sequences based on models and then employ Symbolic PathFinder to perform symbolic execution on each sequence. Similarly, Jensen et al. [63] used concolic execution to derive event sequences of an Android app. But their focus is different from the former works, which is covering the specific paths that can lead to a specific target state in an Android app.

9.3 Dealing with Implicit Callbacks and Dynamic Class Loading

Recently, a number of researchers focus on static analysis for the security and performance of Android applications. However, the component-based and event-driven nature of Android framework presents many challenges for static analysis. One of the challenges is that the implicit callback mechanism provided and orchestrated by the Android framework makes the generation of precise control flow graph (the essential part of program analysis) very difficult.

Cao et al. [64] implemented a tool, called EdgeMiner, which can statically analyze the entire Android framework to generate API summaries that describe implicit control flow transitions through the Android framework. Totally, they found 19,647 callbacks from Android framework, which can be used to augment the existing static analysis tools for Android apps. Actually, the goal of the CBG we proposed in this paper is similar to Cao's work. We can use their results to reconstruct our CBG more precisely.

Yang et al. [65] considered user-event-driven components and the related sequences of callbacks from the Android framework to the application code, both for lifecycle callbacks and for event handler callbacks. They proposed a context-sensitive static analysis method to capture such callback methods and generate a precise callback control-flow graph. In their work, they use an existing analysis method [66] of GUI-related objects to assist their static analysis.

Apart from the implicit callbacks, static analysis of Android apps also faces the challenges inherited from Java, one of which is dynamic class loading. The key point for dynamic class loading is how a static analysis tool knows which calls the program will execute. Bodden et al. [67] presented a tool chain called TamiFlex to aid static analysis for dealing with the reflection and custom class loaders. They designed and implemented two Java instrumentation agents that can emit all the dynamically loaded classes into a repository and log the reflective method calls. With such information, they can insert the offline-transformed classes into the program, then traditional static analysis techniques can work on the modified codes. In the future, we will use this technique to improve Relda2 for handling dynamic class loading.

9.4 Privacy Leak Detection

Many researchers use program analysis to detect potential bugs in Android apps, like privacy leaks.

Kim et al. [20] applied the abstract interpretation framework to detect privacy leaks in Android apps. Gibler et al. [13] applied the traditional static analysis technique to detect privacy leaks in Android apps. However, their experimental results show that the performance of their method does not scale up. FlowDroid [15], proposed by Arzt et al., is a highly-precise static taint analysis tool for Android apps. It contains a precise model of Android lifecycle, that can properly handle callbacks invoked by the Android framework. It is a context-, flow-, field-, object-sensitive taint analysis tool. Unfortunately, it is computation- and memory-intensive. Huang et al. [68] designed a tool, called DroidInfer to detect privacy leaks in Android apps. They proposed an approach using type-based taint analysis and it explains source-sink flows intuitively in terms of CFL-reachability paths. Here CFL stands for Context Free Language.

Besides static analysis, many approaches for analyzing apps are based on dynamic analysis. For instance, Enck et al. [69] implemented a dynamic analysis tool, called Taintdroid, to detect privacy leaks on smartphones. However, it tends to be manually driven, that is, relevant program inputs need to be generated manually. This may be quite time-consuming, yet the coverage might be low.

Different from resource leaks, the privacy leak problem focuses on the flow or propagation of privacy data, that is, it needs to trace the variables that carry privacy data and detect whether the data can be sent to some unexpected place. Currently, the state-of-the-art solution for detecting privacy leaks is taint analysis that is a specific data flow analysis technique. On the other hand, the goals of privacy leak detection and resource leak detection are both to find the mismatched or matched behaviors in some program paths. So the tools for privacy leaks (e.g., FlowDroid) could be extended to detect resource leaks with the assistant of the resource table. However, our approach contains more specific optimizations for the resource leak problem such as VFG and the multi-thread technique, that can greatly improve the analysis efficiency.

9.5 Typestate Related Analysis

Typestate [70] is an elegant programming language concept for specifying a class of temporal safety properties. Specifically, it can encode correct usage rules for many common libraries and application programming interfaces (APIs). Our flow-sensitive approach (Section 6) can be regarded as a typestate related approach. We encode the resource leak bugs as temporal safety properties expressed in CTL.

Fink et al. [71] presented a composite and staged typestate verification system to support strong updates and more precise alias analysis for Java programs. It checks typestate properties by solving a flow-sensitive, context-sensitive dataflow problem on a combined domain of typestate and pointer information. Bierhoff and Aldrich [72] proposed a sound modular protocol checking approach, based on typestates to guarantee the absence of protocol violations at runtime. They defined a new abstraction (access permissions), which contains typestate and object aliasing information, to assist developers express their protocol design intent. Their approach keeps track of the degree of possible aliasing of each object reference. Aldrich et al. [73], [74] also proposed typestate-oriented programming, which is a novel paradigm that can introduce states to describe object

interfaces and representation. By integrating states into the language, they can enhance the expressive power of the language while minimizing added complexity.

10 Conclusion

This paper makes a comprehensive analysis of resource leak problems in Android apps. We give the definition and the categories for resources in Android apps. We collect a resource table which includes the resources that are required to release manually. Based on the resource table, we propose a general process to detect resource leaks in Android apps automatically. We implement our approach in the tool Relda2, and evaluate it with 103 real-world Android apps. Our experimental results confirm that Relda2 is effective and practical to detect resource leaks in non-trival Android apps.

Compared with our previous work [1], we significantly extend it in six aspects: (1) constructing a CBG to handle callbacks invoked by Android framework; (2) improving the FCG construction algorithm to support more precise interprocedural analysis via the callback graph; (3) proposing a flow-sensitive method to detect resource leaks, thus eliminating some false negatives in the flow-insensitive approach; (4) accelerating the analysis efficiency with multi-threading technique; (5) enhancing our evaluation with more real-world apps; (6) developing an instrumentation tool *AddStake* that can be combined with error-guessing testing to confirm whether the resource leaks reported by Relda2 are real bugs.

However, there are still some points for possible improvements. The size and complexity of Android apps are continuously increasing and lots of new features are added into the Android system; thus we need some elaborately designed techniques to support the new Android specifics and effective analysis. On the other hand, without the significant reduction of the efficiency, we will include more accurate analysis techniques (such as symbolic execution or dynamic analysis) in our method to eliminate the false positives as much as possible. At last, we may extend the proposed static analysis process to other similar problems (e.g., privacy-related ones) of Android apps in the future.

Acknowledgments

The authors would like to thank the editor and anonymous reviewers for their constructive comments and suggestions that greatly improve their article. They also sincerely thank Bin Liang at Renmin University of China and Bo Jiang at Beihang University, for their suggestions. This work is supported by the National Basic Research (973) Program of China under Grant No. 2014CB340701, and the National Natural Science Foundation of China under Grant No. 91418206.This paper is a completely revised and extended version of a paper [1] presented at the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE'13). This work was done while the authors are/were with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. Jun Yan is the corresponding author.

References

[1] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. (2013). Characterizing and detecting resource leaks in Android applications. in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, pp. 389–398. [Online]. Available: http://dx.doi.org/10.1109/ASE.2013.6693097

[2] (2015). IDC. smartphone os market share, 2015, 2014, 2013 and 2012 [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[3] A. Pathak, Y. C. Hu, and M. Zhang. (2012). Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. in *Proc. 7th Eur. Conf. Comput. Syst,*. pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168841

[4] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. (2010). Accurate online power estimation and automatic battery behavior based power model generation for smartphones. in *Proc. 8th Int. Conf. Hardware/Software Codesign Syst. Synthesis.*, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/1878961.1878982

[5] A. Shye, B. Scholbrock, and G. Memik. (2009). Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. in *Proc. 42st Annu. IEEE/ACM Int. Symp. Microarchit.*, pp. 168–178. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669135

[6] T. McDonnell, B. Ray, and M. Kim. (2013). An empirical study of API stability and adoption in the Android ecosystem. in *Proc. IEEE Int. Conf. Softw. Maintenance*, pp. 70–79. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2013.18

[7] C. Hu and I. Neamtiu. (2011). Automating GUI testing for Android applications. in *Proc. 6th Int. Workshop Autom. Softw. Test*, pp. 77–83. [Online]. Available: http://doi.acm.org/10.1145/1982595.1982612

[8] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. (2012). A whitebox approach for automated security testing of Android applications on the cloud. in *Proc. 7th Int. Workshop Autom. Softw. Test*, pp. 22–28. [Online]. Available: http://dx.doi.org/10.1109/IWAST.2012.6228986

[9] A. Machiry, R. Tahiliani, and M. Naik. (2013). Dynodroid: An input generation system for Android apps. in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[10] D. Yan, S. Yang, and A. Rountev. (2013). Systematic testing for resource leaks in Android applications. in *Proc. 24th IEEE Int. Symp. Softw. Rel. Eng.*, pp. 411–420. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2013.6698894

[11] S. Anand, M. Naik, M. J. Harrold, and H. Yang. (2012). Automated concolic testing of smartphone apps. in *Proc. 20th ACM SIGSOFT Symp. Found. Softw. Eng.*, pp. 59:1–59:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

[12] N. Mirzaei, S. Malek, C. S. Pasareanu, N. Esfahani, and R. Mahmood. (2012). Testing Android apps through symbolic execution. *ACM SIGSOFT Softw. Eng. Notes*. [Online]. vol. 37, no. 6, pp. 1–5. Available: http://doi.acm.org/10.1145/2382756.2382798

[13] C. Gibler, J. Crussell, J. Erickson, and H. Chen. (2012). Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale. in *Proc. 5th Int. Conf. Trust Trustworthy Comput.*, pp. 291–307. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30921-2_17

[14] É. Payet, and F. Spoto. (2012). Static analysis of Android programs. *Inform. Softw. Technol.* [Online]. vol. 54, no. 11, pp. 1192–1201. Available: http://dx.doi.org/10.1016/j.infsof.2012.05.003

[15] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, p. 29. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[16] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. (2014). Asdroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. in *Proc. 36th Int. Conf. Softw. Eng.*, pp. 1036–1046. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568301

[17] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. (2012). What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. in *Proc. 10th Int. Conf. Mobile Syst., Appl. Serv.*, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307661

[18] (2013). Package index | Android developer [Online]. Available: http://developer.android.com/reference/packages.html

[19] (2013). Tools to work with Android .dex and java .class files - google project hosting [Online]. Available: http://code.google.com/p/dex2jar/

[20] J. Kim, Y. Yoon, K. Yi, and J. Shin, "Scandal: Static analyzer for detecting privacy leaks in Android applications," in *Proc. Mobile Security Technol.*, 2012, pp. 1–6.

[21] A. Bartel, J. Klein, Y. L. Traon, and M. Monperrus. (2012). Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. in *Proc. ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2259051.2259056

[22] (2014). Androguard [Online]. Available: http://code.google.com/p/androguard/

[23] (2014). Google Play market [Online]. Available: https://play.google.com/store/apps

[24] (2014). Wandoujia [Online]. Available: http://www.wandoujia.com/apps/

[25] (2015). F-droid | free and open source android app repository [Online]. Available: https://f-droid.org/

[26] (2015). Error guessing [Online]. Available: https://en.wikipedia.org/wiki/Error_guessing

[27] (2013). Camera resource hasn't been released properly, other application fail to connect to camera [Online]. Available: http://stackoverflow.com/question/2563973/android-fail-to-connect-to-ca mera

[28] (2013). Registered sensor hasn't been released in baidumap [Online]. Available: http://tieba.baidu.com/p/1364806960

[29] (2013). Camera API [Online]. Available: http://developer.android.com/reference/android/hardware/Camera.html

[30] (2013). Mediaplayer API [Online]. Available: http://developer.android.com/reference/android/media/MediaPlayer.html

[31] (2013). Sensormanager API [Online]. Available: http://developer.android.com/reference/android/hardware/SensorManager.h tml

[32] (2015). Call graph [Online]. Available: https://en.wikipedia.org/wiki/Call_graph

[33] B. Steffen, J. Knoop, and O. Rüthing. (1990). The value flow graph: A program representation for optimal program transformations. in *Proc. Eur. Symp. Program.*, vol. 432, pp. 389–405. [Online]. Available: http://dx.doi.org/10.1007/3-540-52592-0_76

[34] S. Cherem, L. Princehouse, and R. Rugina. (2007). Practical memory leak detection using guarded value-flow analysis. in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, pp. 480–491. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250789

[35] Y. Sui, D. Ye, and J. Xue. (2012). Static memory leak detection using full-sparse value-flow analysis. in *Proc. Int. Symp. Softw. Testing Anal.*, pp. 254–264. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336784

[36] E. M. Clarke, E. A. Emerson, and A. P. Sistla. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* [Online]. vol. 8, no. 2, pp. 244–263. [Online]. Available: http://doi.acm.org/10.1145/5397.5399

[37] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. (1990). Symbolic model checking: 10^{20} states and beyond. in *Proc. 5th Annu. Symp. Logic Comput. Sci.*, pp. 428–439. [Online]. Available: http://dx.doi.org/10.1109/LICS.1990.113767

[38] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. (2002). Nusmv 2: An opensource tool for symbolic model checking. in *Proc. Int. Conf. Comput. Aided Verification*, vol. 2404, pp. 359–364. [Online]. Available: http://dx.doi.org/10.1007/3-540-45657-0_29

[39] (2013). Jd-gui [Online]. Available: http://java.decompiler.free.fr/?q=jdgui

[40] T. J. McCabe. (1976, Dec.). A complexity measure. *IEEE Tran. Softw. Eng.* [Online] vol. 2, no. 4, pp. 308–320. Available: http://dx.doi.org/10.1109/TSE.1976.233837

[41] (2016). Ui/application exerciser monkey | android developers [Online]. Available: http://developer.Android.com/tools/help/monkey.html

[42] (2016). Managed code [Online]. Available: https://en.wikipedia.org/wiki/Managed_code

[43] N. Mitchell and G. Sevitsky. (2003). Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. in *Proc. 17th Eur. Conf. Object-Oriented Program.*, pp. 351–377. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45070-2_16

[44] G. H. Xu and A. Rountev. (2008). Precise memory leak detection for Java software using container profiling. in *Proc. 30th Int. Conf. Softw. Eng.*, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368110

[45] G. H. Xu, M. D. Bond, F. Qin, and A. Rountev. (2011). Leakchaser: Helping programmers narrow down causes of memory leaks. in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Design Implementation.*, pp. 270–282. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993530

[46] D. Yan, G. H. Xu, S. Yang, and A. Rountev. (2014). Leakchecker: Practical static memory leak detection for managed languages. in *Proc. 12th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, p. 87. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544151

[47] M. D. Bond and K. S. McKinley. (2009). Leak pruning. in *Proc. 14th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, pp. 277–288. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508277

[48] M. Arnold, M. T. Vechev, and E. Yahav. (2008). QVM: An efficient runtime for detecting defects in deployed systems. in *Proc. 23rd Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang. Appl.*, pp. 143–162. [Online]. Available: http://doi.acm.org/10.1145/1449764.1449776

[49] M. Arnold, M. T. Vechev, and E. Yahav. (2011). QVM: An efficient runtime for detecting defects in deployed systems. *ACM Trans. Softw. Eng. Methodol.* [Online]. vol. 21, no. 1, p. 2. [Online]. Available: http://doi.acm.org/10.1145/2063239.2063241

[50] E. Torlak and S. Chandra. (2010). Effective interprocedural resource leak detection. in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, pp. 535–544. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806876

[51] W. Weimer and G. C. Necula. (2008). Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.* [Online]. vol. 30, no. 2. [Online]. Available: http://doi.acm.org/10.1145/1330017.1330019

[52] Y. Liu, C. Xu, and S. C. Cheung. (2013). Where has my battery gone? finding sensor related energy black holes in smartphone applications. in *Proc. IEEE Int. Conf. Pervasive Comput. Commun.*, pp. 2–10. [Online]. Available: http://dx.doi.org/10.1109/PerCom.2013.6526708

[53] Y. Liu, C. Xu, S. Cheung, and J. Lu. (2014, Sep.). Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Trans. Softw. Eng.* [Online]. vol. 40, no. 9, pp. 911–940. [Online]. Available: http://dx.doi.org/10.1109/TSE.2014.2323982

[54] W. Visser, K. Havelund, G. P. Brat, and S. Park. (2000). Model checking programs. in *Proc. 15th IEEE Int. Conf. Automated Softw.*, pp. 3–12. [Online]. Available: http://dx.doi.org/10.1109/ASE.2000.873645

[55] (2015). Data-flow analysis [Online]. Available: https://en.wikipedia.org/wiki/Data-flow_analysis

[56] K. Kim and H. Cha. (2013). Wakescope: Runtime wakelock anomaly management scheme for Android platform. in *Proc. Int. Conf. Embedded Softw.*, pp. 27:1–27:10. [Online]. Available: http://dx.doi.org/10.1109/EMSOFT.2013.6658605

[57] P. Runeson, M. Alexandersson, and O. Nyholm. (2007). Detection of duplicate defect reports using natural language processing. in *Proc. 29th Int. Conf. Softw. Eng.*, pp. 499–510. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.32

[58] T. Azim and I. Neamtiu. (2013). Targeted and depth-first exploration for systematic testing of Android apps. in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program., Syst., Lang., Appl.*, pp. 641–660. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509549

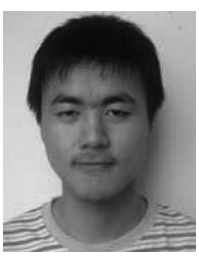
[59] W. Yang, M. R. Prasad, and T. Xie. (2013). A grey-box approach for automated gui-model generation of mobile applications. in *Proc. 16th Int. Conf. Fundam. Approaches Softw. Eng.*, pp. 250–265. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37057-1_19

[60] F. Gross, G. Fraser, and A. Zeller. (2012). Search-based system testing: high coverage, no false alarms. in *Proc. Int. Symp. Softw. Testing Anal.*, pp. 67–77. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336762

[61] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon. (2012). Using GUI ripping for automated testing of Android applications. in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, pp. 258–261. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351717

[62] (2016). Gui ripper wiki [Online]. Available: http://wpage.unina.it/ptramont/GUIRipperWiki.htm

[63] C. S. Jensen, M. R. Prasad, and A. Møller. (2013). Automated testing with targeted event sequence generation. in *Proc. Int. Symp. Softw. Testing Anal.*, pp. 67–77. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483777
[64] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. (2015). EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework. in *Proc. 22nd Annu. Netw. Distrib. Syst. Security Symp.* [Online]. Available: http://www.internetsociety.org/doc/edgeminer-automatically-detecting-im plicit-control-flow-transitions-through-android-framework
[65] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. (2015). Static control-flow analysis of user-driven callbacks in Android applications. in *Proc. 37th IEEE/ACM Int. Conf. Softw. Eng.*, pp. 89–99. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.31
[66] A. Rountev and D. Yan. (2014). Static reference analysis for GUI objects in Android software. in *Proc. 12th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, pp. 143–153. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544159
[67] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. (2011). Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. in *Proc. 33rd Int. Conf. Softw. Eng.*, pp. 241–250. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985827
[68] W. Huang, Y. Dong, A. Milanova, and J. Dolby. (2015). Scalable and precise taint analysis for Android. in *Proc. Int. Symp. Softw. Testing*, pp. 106–117. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771803
[69] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. (2010). Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. in *Proc. 9th USENIX Symp. Operating Syst. Design Implementation*, pp. 393–407. [Online]. Available: http://www.usenix.org/events/osdi10/tech/full_papers/Enck.pdf
[70] R. E. Strom and S. Yemini. (1986, Jan.). Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.* [Online]. vol. 12, no. 1, pp. 157–171. [Online]. Available: http://dx.doi.org/10.1109/TSE.1986.6312929
[71] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. (2006). Effective typestate verification in the presence of aliasing. in *Proc. ACM/SIGSOFT Int. Symp. Softw. Testing Anal.*, pp. 133–144. [Online]. Available: http://doi.acm.org/10.1145/1146238.1146254
[72] K. Bierhoff and J. Aldrich. (2007). Modular typestate checking of aliased objects. in *Proc. 22nd Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl.*, pp. 301–320. [Online]. Available: http://doi.acm.org/10.1145/1297027.1297050
[73] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. (2009). Typestate-oriented programming. in *Proc. 24th Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl.*, pp. 1015–1022. [Online]. Available: http://doi.acm.org/10.1145/1639950.1640073
[74] R. Garcia, É. Tanter, R. Wolff, and J. Aldrich. (2014). Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.* [Online]. vol. 36, no. 4, pp. 12:1–12:44. Available: http://doi.acm.org/10.1145/2629609



Tianyong Wu received the BS and MS degrees from Xiamen University in 2011 and the University of Chinese Academy of Sciences (UCAS) in 2014, respectively. He is currently working toward the PhD degree at the University of Chinese Academy of Sciences. His research interests include software test generation and static analysis.

Jierui Liu received the BS degree from the Harbin Institute of Technology in 2013. He is currently working toward the master's degree at the University of Chinese Academy of Sciences. His research interests include static analysis and Android application testing.

Zhenbo Xu received the BS and PhD degrees from the University of Science and Technology of China in 2009 and 2015, respectively. He visited the Institute of Software, Chinese Academy of Sciences from 2010 to 2015. His research interests include static analysis and bug finding.

Chaorong Guo received the BS and MS degrees from Beihang University in 2011 and the University of Chinese Academy of Sciences in 2014, respectively. Her research interests include static analysis on Android apps.

Yanli Zhang received the BS and MS degrees from JiLin University in 2012 and the University of Chinese Academy of Sciences, respectively. Her research interests include static analysis and Android application testing.

Jun Yan received the BS degree in EE from the University of Science and Technology of China in 2001 and the PhD degree from the Graduate University, Chinese Academy of Science in 2007, respectively. He is now an associate research professor at the Institute of Software, Chinese Academy of Sciences. His research interests include program analysis and software testing, constraint processing.

Jian Zhang is a research professor at the Institute of Software, Chinese Academy of Sciences (ISCAS). His main research interests include automated reasoning, constraint satisfaction, source code analysis, and software testing. He has served on the program committee of about 60 international conferences. He also serves on the editorial boards of several journals including *Frontiers of Computer Science*, *Journal of Computer Science and Technology*. He is a senior member of ACM, the IEEE, and China Computer Federation (CCF).