LeakDAF: An Automated Tool for Detecting Leaked Activities and Fragments of Android Applications

MA Jun*† LIU Sheng*† YUE Shengtao*† TAO Xianping*† and LU Jian*†
*State Key Laboratory for Novel Software Technology, Nanjing University
†Department of Computer Science and Technology, Nanjing University
NO.22 Hankou Road, Nanjing, China, 210093
Email: majun@nju.edu.cn, 121220058@snju.edu.cn, mg1533079@smail.nju.edu.cn, txp@nju.edu.cn, lj@nju.edu.cn

***Abstract*—Memory leak, one of the most common problems threatening android apps, might drain the limited memory of mobile devices, cause unexpected delays, no-responses or even crashes to apps. *Activity/Fragment Leak* is one of the most common and serious causes of memory leaks and has a vast influence on the Android app market. Existing work to identify leaked activities/fragments either depend highly on the experience of developers, or require app's source code and manual interactions. In this paper, we propose an automatic tool named *LeakDAF* for detecting leaked activities/fragments automatically without manual intervention. LeakDAF makes use of UI testing technique to execute automatically the app under test, and applies memory analysis technique to inspect dumped heap files to identify leaked activities and fragments based on Android's mechanisms for managing them. To evaluate the effectiveness of LeakDAF, we successfully applied it to 35 open source and 64 commercial apps, and we detected at least one leaked activity or fragment for 10 open source apps and 35 commercial apps.**

***Keywords*-Android; testing; memory leak; automated leak detection**

I. Introduction

Nowadays, mobile smart phones have been becoming necessary parts of our everyday life. Millions of apps provide users the possibilities and flexibilities of enjoying a great variety of services anytime anywhere. Meanwhile, apps, if not carefully designed and implemented, would also bring some problems to users. Among these problems, memory leak is one of the most commons, which might drain the limited memory of mobile devices, cause unexpected delays, no-responses or even crashes to apps.

There are many reasons for memory leak, such as leaked activities/fragments, forgetting to recycle `Bitmap` instances, forgetting to unregister given event listeners, forgetting to close `Cursor` instances after accessing database, etc. Among these, *Activity/Fragment Leak* is one of the most common and serious causes and has a vast influence on the Android app market. A destroyed activity/fragment is leaked if it is still unexpected referenced by some other objects and can not be released by the GC.

A typical way to detect memory leak is usually done manually in an ad-hoc way. A develop (or a tester) runs the target app and monitors the memory status of the app while execution; if the memory used by the app grows unexpectedly or an Out-of-Memory(OOM) error is thrown, the developer dumps the heap and further analyzes the dumped `.hprof` file via memory analysis tools(e.g., MAT[1]). Whether the cause of leak can be detected and how long the process would take are highly dependent on the experience of the developer. The open source project LeakCanary[2] provides a memory leak detection library for Android. LeakCanary will automatically show a notification when an activity memory leak is detected in the debug build of an app during the execution of the app. However, source code and manual interactions are still required by LeakCanary to drive the execution of application.

To address the challenges faced by existing leak detection techniques, we propose a new tool, named "*LeakDAF*", for detecting leaked activities/fragments in an Android app automatically. Briefly speaking, observing the life-cycle of Android's activities and fragments, and the mechanisms for managing them, LeakDAF combines both UI testing (to drive app automatically) and memory analysis techniques (to analyze dumped `.hprof` files to detect leaked activities/fragments). The main contributions of this paper are:

- We combine both UI testing and memory analysis techniques and propose an automated tool, LeakDAF, for detecting leaked activities/fragments of Android apps;
- LeakDAF requires neither app modification nor manual interaction to execute apps, which greatly extends its application domain;
- We successfully applied LeakDAF to 35 open source apps and 64 commercial apps, and found 10 (28.6%) open source apps and 35(54.7%) commercial apps having at least one detected leaked activity or fragment.

The rest of this paper is organized as follows: Sec.II gives a brief introduction to the background of this work; Sec.III reveals the key ideas and structures of the LeakDAF tool; Sec.IV depicts the implementation details of our prototype of LeakDAF and demonstrates the experiments carried out by applying our prototype to both open-source apps and commercial apps; after briefly introducing some related work in Sec.V, we finally conclude this paper and discuss our future work in Sec.VI.

II. Background

A. Garbage Collection in Android DVM

An Android app is generally written in Java and runs on a special Java virtual machine, the Android Dalvik Virtual Machine(DVM) with its own process and heap. Each running app has its own Garbage Collector to automatically free garbage objects. A Dalvik's Garbage Collector uses *Mark and Sweep* approach[3], which traces out the entire collection of objects that are *directly* or *indirectly* accessible by the program. The objects that a program can access directly are called the *roots*. There are many types of roots, such as classes loaded by system class loader, live threads, and objects referenced by local variables on the stack as well as by any static variables, etc. Meanwhile, an object is indirectly accessible if it is referenced by some other (directly or indirectly) accessible object. All accessible objects are said to be *live*, while others are *garbage* and should be freed up.

If an object, believed to be useless and a garbage to be freed up (from the point of a developer), is unexpectedly referenced by one or more other live object(s), the object can not be marked as garbage by GC; thus, memories retained by the object would not be released. We call such useless but live objects "*leaked objects*".

B. Activity and Fragment

Activities are one of the basic components for building Android apps. Generally speaking, an activity represents a single screen with a user interface. An app may consist of multiple activities. For example, a SMS app might have one activity that shows a list of new SMSs, another activity to compose a SMS, and another activity for reading SMSs. Although these activities work together to form a cohesive user experience in the app, they are loosely coupled or even independent. An activity class is implemented as a subclass of `android.app.Activity` which declares a set of life-cycle callback methods (e.g., `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, `onDestroy()`, etc.). These methods are called by the underlying Android framework, and can be overridden for specifying unique behaviors when the activity is created, started, resumed, paused, stopped or destroyed accordingly.

Generally, an activity would be destroyed once it is popped up from its corresponding *back stack*; while the popping ups usually take place when an user presses the "Back" button or rotates the screen.

Fragments were introduced in Android (since API level 11) to support more dynamic and flexible UI designs on large screens, such as tablets. A Fragment is a piece of an app's user interface or behavior that can be placed in an Activity. Multiple fragments can be combined in a single activity to build a multi-pane UI and a fragment can be reused in multiple activities. More importantly, *a fragment must always be embedded in an activity and the fragment's lifecycle is directly affected by the host activity's lifecycle*. If an activity is stopped, no fragments inside it can be started; and, if the activity is destroyed, all fragments will be destroyed as well.

A fragment class is implemented as a subclass of `android.app.Fragment` (or `android.support.v4.app.Fragment` for old versions) which declares a bunch of life-cycle callback methods (e.g., `onCreateView()`, `onAtach()`, `onDetach()`, `onDestroyed()`, etc.) to be called accordingly by the Android framework when the status of a fragment changes.

C. Leaked Activities/Fragments

The complex life-cycle management mechanisms of Android make leaks of activities and fragments very common. Generally, the life cycle of an activity starts when it is created and ends when it is destroyed, and the activity should be released by GC (not necessarily immediately) after it is destroyed. However, there are many cases where an activity is destroyed but cannot be freed up, as it is unexpectedly referenced by some objects with longer life cycle.

For example, if a non-static inner (possibly anonymous) class (or an instance of it) defined within an activity class, is referenced somewhere when an activity is destroyed, the activity cannot be released as the inner class holds a strong reference to it. Besides, as the life cycle of a static referenced object can be as long as that of the app, an activity is likely becoming leaked with high probability if it is (directly or indirectly) referenced by a static field defined in one or more classes.

Situations mentioned above may happen as well when a fragment is destroyed. However, a destroyed fragment may be reused by the app for the purpose of, for example, improving the app's responsiveness.

D. Leak Example

Lock-Pattern-Generator[4] is a simple program with over 5,000,000 downloads that generates a random lock screen pattern and helps user memorize it before applying it. `GeneratorActivity` is the main activity of the app. As shown in List.1, the developer tries to handle out of memory errors gracefully by defining an anonymous inner `UncaughtExceptionHandler` class. However, each time a `GeneratorActivity` is created, an `UncaughtExceptionHandler` instance would be created as well, which holds reference to the original `defaultHandler` for the purpose of calling it for default handling (line 15). Thus a chain of multiple instances of the non-static inner `UncaughtExceptionHandler` class would be formed. As the type of each instance in the chain is the anonymous inner `UncaughtExceptionHandler` class, it holds a reference to its associated instance of `GeneratorActivity`, resulting that the associated activity cannot be freed either.

```
1  public class GeneratorActivity extends BaseActivity{
2      public void onCreate(Bundle state ){
3          ...
4          // set a default exception handler to catch out of memory errors
5          // gracefully
6          final Thread.UncaughtExceptionHandler exceptionHandler =
7              Thread.getDefaultUncaughtExceptionHandler();
8          Thread.setDefaultUncaughtExceptionHandler(
9                  new Thread.UncaughtExceptionHandler() {
10             @Override
11             public void uncaughtException(Thread thread, Throwable
               throwable) {
12                 ...
13                 // punt if it's not an exception we can handle
14                 exceptionHandler.uncaughtException(thread, throwable);
15             }
16         });
17         ...
18     }
19     ...
20 }
```

Listing 1. Leaked Activity in in.shick.lockpatterngenerator[4]

III. Detecting Leaked Activities/Fragments

Considering the complex life-cycle management mechanisms provided by the Android platform, it is not an easy job for developer to discover such leaks. Therefore, providing a tool that could detect leaked activities/fragments automatically would help developers a lot.

A. Overview of LeakDAF

We designed and implemented LeakDAF, which makes use of UI testing technique to execute automatically the app under test, and applies memory analysis technique to inspect dumped heap files to identify leaked activities and fragments.

As shown in Figure.1, LeakDAF consists of two main components: (a) Ui-Explorer (UE) and (b) Leak Analyzer(LA). For each app (packaged in a `.apk` file), (step 1) LeakDAF first installs and starts the app on a target device(either an actual device or an emulator). Then, (step 2) UE calculates a set of all possible candidate actions/events by analyzing the status of the current screen, selects one action/event from the set based on given strategy and further triggers the action/event. If the app crashes during execution, LeakDAF dumps the heap of the app and restarts it. Step 2 repeats until the number of actions performed exceeds predefined threshold, after which (step 3) LA would dump the heap and analyzes all dumped `.hprof` files to find out leaked activities and (quasi-)leaked fragments.

To detecting leaked activities and fragments automatically, we have to address following 2 questions:

- **Q1**: *How to trigger leakages of activities/fragments?*
- **Q2**: *How to identify leaked activities/fragments?*

B. UiExplorer

Q1 is addressed by UiExplorer(UE), which is responsible for generating proper actions that trigger execution of the target app. As mentioned, an activity/fragment leaks only if it is destroyed and a fragment's life cycle depends highly on that of its hosting activity; therefore, we pay special attention to events/actions which would destroy activities with high probabilities and two types of events are taken into consideration: (a) ***Back***: a focused activity would generally be popped up from the back stack and destroyed if the "back" button is pressed; (b) ***Rotation***: in most cases an activity would be destroyed and reconstructed when the screen is rotated[1]. With these in mind, we implemented UE by extending the *Execution-Observation-Selection* cycle of Dynodroid[5], and the overall process of UE is shown in Procedure.1.

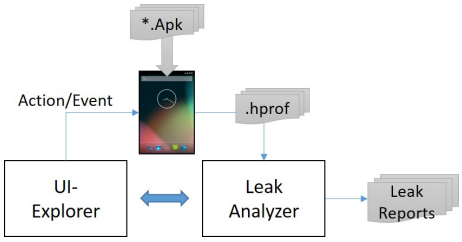


Figure 1. Overview structure of LeakDAF

UE first tries to install and start the app under test (see the initial event at line 1). The execution stage of UE is more complex than that of Dynodroid, as we should destroy activities and fragments for detecting possible leaks. First, UE is designed to check whether a new activity is started and pushed onto the top of the corresponding back stack after executing an event e at line 6. If yes, UE simulates the action of pressing the "back" button to pop up the top activity and return back to previous state (line 13). Then, if it successfully returns back to the previous state, UE would re-trigger the event e again (line 20). Second, after the successful execution of event e, UE further forces the screen to rotate(line 28). If any crash takes place, UE would first notify LA to dump the app's heap (line 8,15,22,30), and then restart the app(line 47,48). Finally, once the number of executed events reaches the predefined threshold number of trials *num_trials*, UE dumps the heap and returns all dumped `.hprof` files stored in L_{heap}.

We believe that switches between activities are more likely triggered by UI events than by system events. Therefore, in the design of UE, we ignore all system events during the stage of observation and selection, although they can be easily added.

C. Leak Analyzer

Leak Analyzer(LA) is responsible for **Q2**. The overall procedure of LA is shown in Procedure.2. For each dumped `.hprof` file f_{heap}, LA tries to find all leaked activities and fragments by calling `findAllLeakedInstances()`. The procedure `findAllLeakedInstances()` (see Procedure.3) first finds the set of all activities and fragments (line 2,3)[2]. Then, each activity or fragment is further checked

[1]There are cases (e.g., `android:screenOrientation` is fixed to "`landscape`" or "`portrait`" in the AndroidManifest.xml) where an activity would not be destroyed and recreated when rotations take place.

[2]Here the set of all fragments actually should include not only instances of subclasses of `android.app.Fragment` but also those of subclasses of `android.support.v4.app.Fragment`.

Procedure 1 Overall procedure for UiExplorer

```
1:  s← initial status;
2:  e ← event to install and start app under test;
3:  procedure INPUT-GENERATOR(num_trials)
4:      L_heap be an empty list;
5:      for i from 1 to num_trials do
6:          s' ← Executor(e,s);
7:          if s' indicates crash then
8:              DUMPHEAP(L_heap) ;
9:              continue;
10:         end if
11:         if s' represents a new started activity(not the launcher) then
12:             //Press "back"
13:             s'' ← Executor(BACK,s);
14:             if s'' indicates crash then
15:                 DUMPHEAP(L_heap) ;
16:                 continue;
17:             end if
18:             if s'' and s represent the same activity then
19:                 //Re-execute event e
20:                 s' ← Executor(e,s'');
21:                 if s' indicates crash then
22:                     DUMPHEAP(L_heap) ;
23:                     continue;
24:                 end if
25:             end if
26:         end if
27:         //Force the screen to rotate
28:         s ← Executor(ROTATION,s');
29:         if s indicates crash then
30:             DUMPHEAP(L_heap) ;
31:             continue;
32:         end if
33:         //Compute E of all candidate events relevant in current state s
34:         E← Observer(s);
35:         //Select an event e∈E to execute in next iteration
36:         e← Selector(E);
37:     end for
38:     DUMPHEAP(L_heap) ;
39:     return L_heap;
40: end procedure
41:
42: procedure DUMPHEAP(L)
43:     force GC;
44:     //notify LA to dump the heap
45:     f_heap← dump the heap;
46:     append f_heap to L;
47:     s← initial status;
48:     e ← event to install and start app under test;
49: end procedure
```

Procedure 2 Overall procedure for Leak Analyzer

```
1:  procedure LEAK-ANALYZER(L_heap)
2:      L_act ← empty list;
3:      L_fgmt ← empty list;
4:      for each f_heap ∈ L_heap do
5:          ⟨L'_act,L'_fgmt⟩ ← FINDALLLEAKEDINSTANCES(f_heap);
6:          L_act ← L_act ∪ L'_act;
7:          L_fgmt ← L_fgmt ∪ L'_fgmt;
8:      end for
9:      return ⟨L_act,L_fgmt⟩;
10: end procedure
```

by the `isDestroyed()` and `isLeaked()` procedures, and is added to L_{act} or L_{fgmt} accordingly if it is determined as leaked(line 6-15).

Each activity maintains a boolean field `mDestroyed` indicating whether it is successfully destroyed. So to determine whether an activity is destroyed or not, `isDestroyed()` simply returns the value of its `mDestroyed` field (line 20).

Each fragment also maintains a series of fields

Procedure 3 Procedure for finding all leaked instances

```
1:  procedure FINDALLLEAKEDINSTANCES(f_heap)
2:      L_act_candidate ← GETALLACTIVITIES(f_heap);
3:      L_fgmt_canditate ← GETALLFRAGMENTS(f_heap);
4:      L_act ← empty list;
5:      L_fgmt ← empty list;
6:      for each act ∈ L_act_candidate do
7:          if ISDESTROYED(act)&&ISLEAKED(act) then
8:              append act to L_act;
9:          end if
10:     end for
11:     for each fgmt ∈ L_fgmt_canditate do
12:         if ISDESTROYED(fgmt)&&ISLEAKED(fgmt) then
13:             append fgmt to L_fgmt;
14:         end if
15:     end for
16:     return ⟨L_act,L_fgmt⟩;
17: end procedure
18:
19: procedure ISDESTROYED(obj)
20:     if obj is an activity then return obj.mDestroyed;
21:     else// obj is a fragment
22:         if all fields in Table.I of obj are reset&& obj is not managed
    by any non-destroyed FragmentManager then
23:             return true;
24:         end if
25:         return false;
26:     end if
27: end procedure
28:
29: procedure ISLEAKED(obj)
30:     L_P← GETALLPATHSTOGCROOT(obj);
31:     L_P← FILTERPATHS(L_P);
32:     if L_P is empty then
33:         return false;
34:     end if
35:     return true;
36: end procedure
```

(e.g., `mIndex`, `mAdded`, `mRetaining`, etc.) indicating its runtime state. Once a hosting activity is destroyed, Android would retain its embedded fragments with `mRetaining=true` and should not be destroyed; while, Android would reset the states of the embedded fragments with `mRetaining=false` by invoking `initState()` as shown in Table.I.

Table I
CLEARED FRAGMENT STATE BY `INITSTATE()`

Types of Fields					
int		**boolean**		**Object**	
name	value	name	value	name	value
mIndex	-1	mAdded	false	mTag	null
mBackStackNesting	0	mRemoving	false	mWho	null
mFragmentId	0	mResumed	false	mFragmentManager	null
mContainerId	0	mFromLayout	false	mChildFragmentManager	null
—	—	mInLayout	false	mActivity	null
—	—	mRestored	false	mLoaderManager	null
—	—	mHidden	false	—	—
—	—	mDetached	false	—	—
—	—	mRetaining	false	—	—
—	—	mLoadersStarted	false	—	—
—	—	mCheckedForLoaderManager	false	—	—

The basic idea for `isDestroyed()` to determine whether a fragment is destroyed (line 21-26) is to check whether its internal states are reset to the initial values shown in Table.I .

The procedure `isLeaked()` first collects the set of all paths from *obj* to every GC root. Then, `filterPaths()` is called to filter paths that should not be view as leaks (introduced by the app). If any path exists after `filterPaths()`, LA determine *obj* as leaked and we

call each of the remaining path a "*L-path*". `filterPaths()` first excludes all paths consisting of any object of type `FinalizerReference`, `SoftReference`, `WeakReference` or `PhantomReference`. Besides, as leaks may be caused by the Android SDK and there is little a developer can do to fix them, `filterPaths()` also ignores such leaks by excluding paths that match any known leak pattern defined in the `AndroidExcludedRefs.java`[3] provided by LeakCanary.

One thing to be noted, if we detect a destroyed but not freed fragment, we cannot judge it as a real leaked object for sure, unless we know exactly the designed purpose and behavior of the fragment. Without this information, we cannot say that the fragment is really leaked. But, detecting such quasi-leaked fragment is still useful, especially for developers who know the expected functions and behaviors of the fragment. Besides, an app market administrator or an user would benefit as well if he/she wants to estimate the memory usage of an app.

IV. Implementation and Evaluations

In our prototype, UiExplorer is implemented as a single test case of UiAutomator[6], running on client device/emulator for the purpose of speeding up the *Execution-Observation-Selection* cycle. Actions like "KeyPress", "Click", "Rotation", "Sliding", etc., are realized by directly invoking UiAutomator's APIs; while action "LongClick" is achieved by executing a series of "*sendevent*" cmds. Besides, with the API method `dumpWindowHierarchy()` provided by UiAutomator, we could dump the current window's layout hierarchy, from which the location and relevant events of each widget within the current window can be further analyzed. To select the next event to be executed, we simply adopt the `BiasedRandom` strategy of Dynodroid[5]. Leak Analyzer is built on top of AndroMAT[7].

Table II
Emulator Configuration

Feature Name	Feature Value
Device Ram Size	2GB
Sdcard size	2GB
Android API Version	Lollipop (22)
Files on Sdcard	pdf:2, docx:2, txt:2, jpg:2, mp3:2, mp4:2

We evaluated LeakDAF on both open source apps and real-world Android apps. All the experiments were done on a Dell workstation with Win10 as OS, 32GB RAM and Intel Xeon CPU E5-1650 CPU. The Lollipop version (22) of Android, one of the most popular versions installed on 35.5% of all devices[8], was used in our experiments. We created a fresh emulator (with configuration shown in Table.II) for each app, and ran totally 8 emulators simultaneously to speed up the overall running process.

[3]https://github.com/square/leakcanary/blob/master/leakcanary-android/src/main/java/com/squareup/leakcanary/AndroidExcludedRefs.java, accessed Nov.20, 2016

Table III
Statics of Apps(F-Droid)

	Downloads	Apk Size(KB)	#Activity	#Fragment
Max	5,000K∼10,000K	13,742	30	46
Min	10K∼50K	47	1	0
Avg	579K∼1,494K	2,409	8.46	9.46
Median	50K∼100K	1,239	5	3
Total	-	84,314	296	331

A. Open Source Apps

In the first evaluation, we try to show the effectiveness of LeakDAF. We first randomly selected 73 apps from 16 categories F-Droid, then filter out apps that are not available

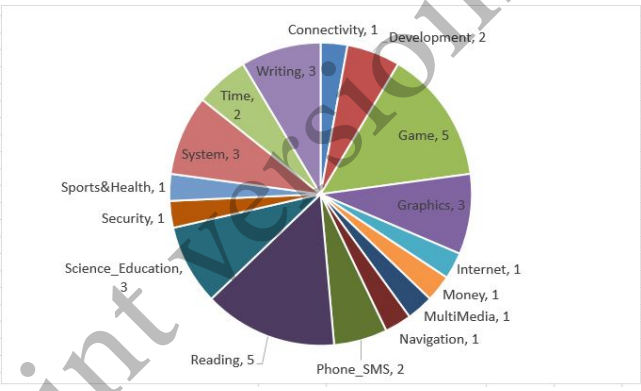


Figure 2. Distribution of F-Droid apps

through Google Play and those with less than 10,000 downloads; finally, only 35 apps left, Figure.2 and Table.III show the statics of these apps.

We downloaded the latest source codes of the 35 apps[4] and enhanced them with LeakCanary 1.5. Then, each app was first executed automatically by LeakDAF (round 1) with 500 events (excluding events for additional back/re-execution and rotations, i.e., line 11∼32 of Procedure.1), and then by Dynodroid[5] (round 2) with 1500 events (system events are enabled and `BiasedRandom` strategy is adopted). Finally, we totally found 29 classes of leaked activities and fragments from 10 apps and Table.IV and Table.V show the statics results.

Thanks to the *back/re-trigger* and *rotation* mechanisms (see Procedure.1), UiExplorer performed better than Dynodroid at triggering more leaked activities/fragments in most cases. Meanwhile, Leak Analyzer(LA) detected many more classes and instances of leaked activities/fragments than LeakCanary did: totally 12 different leaked classes (including 5 classes of activities and 7 classes of fragments) were uniquely detected by LA in the two round experiments. This is caused by the fact that LeakCanary treats each destroyed object(i.e., activity/fragment) separately by starting a background thread each time an (watched) activity/fragment is

[4]Up to Aug. 10, 2016

[5]we re-implemented Dynodroid without instrumenting SDK by using UiAutomator to obtain the window hierarchies, polling `dumpsys` with different parameters like "activity broadcasts", "audio", "location", "telephony.registry", "sensorservice" to update dynamic and system events.

Table IV
DETECTED LEAKED ACTIVITIES/FRAGMENTS CLASSES PER APP (F-DROID)

App	#Class of Leaked Activity				#Class of Leaked Fragment			
	UE-LA	**UE-LC**	**DD-LA**	DD-LC	**UE-LA**	**UE-LC**	**DD-LA**	**DD-LC**
ch.fixme.status[9]	-	1(1)	-	-	-	-	-	-
com.tomatodev.timerdroid[10]	3(1)	2	3	3	4(1)	3	4	4
com.tritop.androsense2[11]	-	1	-	-	-	-	-	-
de.onyxbits.listmyapps[12]	-	1(1)	-	-	-	-	-	-
in.shick.lockpatterngenerator[4]	1	1	-	-	-	-	-	-
org.dmfs.tasks[13]	1(1)	-	-	-	2(2)	-	-	-
com.easytarget.micopi[14]	1(1)	-	-	-	-	-	-	-
org.jtb.alogcat[15]	1	1	1	1	-	-	-	-
org.liberty.android.fantastischmemo[16]	4(3)	1	2(1)	1	7(5)	2	3(2)	1
org.sagemath.droid[17]	1	1	-	1(1)	-	-	-	-
Total	**12(6)**	**8(2)**	**5(1)**	**6(1)**	**13(8)**	**5**	**7(2)**	**5**

* DD: Dynodroid, UE: UiExplorer, LC: LeakCanary, LA: Leak Analyzer
** integers in "()" indicate the numbers of classes uniquely detected by LA or LC with the same explore mechanism(i.e., DD or UE)

Table V
DETECTED LEAKED CLASSES &INSTANCES (F-DROID)

NO	**Class Name**	**Act./ Fgmt.**	**# Detected Instances**			**Status***
			Only by LA	**Only by LC**	**By Both**	
1	ch.fixme.status.Main ⋆	A	-	8		
2	com.tomatodev.timerdroid.activities.ListTimersActivity	A	10	-	6	M
3	com.tomatodev.timerdroid.activities.MainActivity	A	99	-	3	M
4	com.tomatodev.timerdroid.activities.TimerActivity	A	116	-	19	M
5	com.tomatodev.timerdroid.fragments.CategoriesFragment	F	29	-	2	M
6	com.tomatodev.timerdroid.fragments.ListTimersFragment	F	7	-	9	M
7	com.tomatodev.timerdroid.fragments.RunningTimersFragment	F	146	1	5	M
8	com.tomatodev.timerdroid.fragments.TimerFragment	F	108	1	27	M
9	com.tritop.androsense2.MainActivity ⋆	A	-	3	-	
10	de.onyxbits.listmyapps.MainActivity ⋆	A	-	2	-	
11	in.shick.lockpatterngenerator.GeneratorActivity	A	23	-	27	C
12	org.dmfs.tasks.QuickAddDialogFragment	F	3	-	-	C
13	org.dmfs.tasks.TaskListActivity	A	3	-	-	C
14	org.dmfs.tasks.TaskListFragment	F	7	-	-	C
15	org.eztarget.micopi.ui.ContactActivity	A	4	-	-	U
16	org.jtb.alogcat.LogActivity	A	89	5	-	M
17	org.jtb.alogcat.PrefsActivity	A	-	-	2	U
18	org.liberty.android.fantastischmemo.downloader.DownloaderAnyMemo	A	3	-	-	U
19	org.liberty.android.fantastischmemo.downloader.dropbox.DropboxDBListActivity	A	11	-	-	C
20	org.liberty.android.fantastischmemo.downloader.quizlet.QuizletSearchByTitleActivity	A	87	-	-	C
21	org.liberty.android.fantastischmemo.ui.CardFragment	F	9	4	-	C
22	org.liberty.android.fantastischmemo.ui.DownloadTabFragment	F	3	-	-	U
23	org.liberty.android.fantastischmemo.ui.GradeButtonsFragment	F	3	1	-	C
24	org.liberty.android.fantastischmemo.ui.MiscTabFragment	F	3	-	-	C
25	org.liberty.android.fantastischmemo.ui.OpenTabFragment	F	3	-	-	C
26	org.liberty.android.fantastischmemo.ui.RecentListFragment	F	3	-	-	C
27	org.liberty.android.fantastischmemo.ui.StudyActivity	A	3	3	-	C
28	org.liberty.android.fantastischmemo.ui.TwoFieldsCardFragment	F	3	-	-	C
29	org.sagemath.droid.SageActivity	A	1	4	1	M
	Total	**-**	**776**	**24**	**101**	**13**

⋆ Class uniquely detected by LC
* C: Confirmed true leaks; M: Manually checked as true leaks; U: Unsure leaks;

destroyed; and the thread may further dump the heap if necessary. However, a thread in the dumping process would prevent all afterward threads from dumping the heap until it finishes. As a result, a lot of leaked instances are missed.

LeakCanary also reported 3 unique classes (see classes with "⋆" in Table.V) and 24 unique instances of leaked activities or fragments that LA failed to detect. We manually checked the logcat and `.hprof` files and found out that:

- Among the 24 instances uniquely detected by LeakCanary, there were 3 instances whose object IDs cannot be identified in the log traces generated by LeakCanary.
- 17 out the remaining 21 instances were actually not found in the corresponding heaps, indicating they were successfully released.
- Only the last 4 instances (of the same class `org.jtb.alogcat.LogActivity`) remained in the corresponding heap. However, the leak trace reported by LeakCanary for each of the 4 instances did not exist any longer in the corresponding heap; only one path to GC root can be found in heap and the path should be excluded as it matched a known pattern defined in `AndroidExcludedRefs.java`.

The reason may lie in the fact that LeakCanary is an online detector which determines whether an activity/fragment is leaked immediately after it is destroyed; while LA operates in an off-line mode and detects leaked objects by analyzing dumped `.hprof` files. Therefore, LeakCanary is more sensitive to short-term, temporally leaked instances, which cannot be released immediately after the destructions but can be freed up a moment later. Meanwhile, LA ignores such temporally leaked instances but tries to find relatively long-term leaked object in the final dumped heaps.

We have reported the detected leaks to the developers of the corresponding 10 apps, and by the time the paper was submitted, we have received replies from 3 apps confirming that totally 13 reported leaks were real leaks. Specially, 9 out of the 13 confirmed leaks were only detected by LA. For the remaining possible leaked classed detected by LA, we

reviewed the corresponding leak reports as well as the source codes and reran the app manually to see if the leaks can be reproduced. Finally, we found 9 leaked classes (marked as "**M**" in the column **status** of Table.V) with observable causes; while we were unsure of whether the 4 reported leaked classes (marked as "**U**" in the column **status** of Table.V) are real leaks or not.

B. Study on Android Market Apps

In the second experiment, we applied LeakDAF to commercial apps. We randomly downloaded 66 apps with ratings $\geq$ 4.0 from ApkPure[6] on Dec. 27, 2016. We applied LeakDAF to these apps and found only 64 of them can be properly installed and started. Table.VI shows some statics of the 64 apps. Each app was executed automatically by LeakDAF with 500 events (excluding events for additional back/re-execution and rotations).

Table VI
STATICS OF APPS(APKPURE)

	Rating	Downloads	Apk Size(KB)	#Activity	#Fragment
Max	4.90	100,000K∼500,000K	57,882	130	103
Min	4.10	1K∼5K	1,096	3	0
Avg	4.49	14,436K∼60,230K	12,058	29.92	30.47
Median	4.50	1,000K∼5,000K	8,762	17	24
Total	-	-	771,714	1,915	1,950

1) Coverage: In our experiments, LeakDAF successfully visited[7] 839(43.8%) of the total 1,915 types of activities and 587 (30.1%) of the total 1,950 types of fragments. The coverage rates are not high, and the main reasons may be:

- **Complex gestures/inputs:** Some apps rely on complex, app-specific gestures that are not supported by current version of UE.
- **Login required:** Many apps require login to access all activities and fragments. There were 31 apps having at least one activity with a name containing "*login*" or "*auth*".
- **Limited events:** Only 500 events were generated per each app in our experiments, which may be not enough for accessing many activities/fragments, especially for apps with a large number of activities/fragments.
- **App crashes:** In our experiments, 10 apps crashed at least once during execution, and the most unstable app crashed surprisingly 8 times. Crashes forced LeakDAF to restart the apps, and reduced the probabilities to access new activities/fragments.

2) Apps With Leaked Activity/Fragment: To our surprise, we detected 30 (46.9%), 25(39.1%) and 35(54.7%) apps having at least one leaked activity, fragment, or either accordingly as shown in Table.VII. Among these apps, we totally found 62 classes of leaked activity (7.4% of all the visited 839 activity types) and 95 leaked fragment types(16.2% of all the visited 587 fragment types). The biggest numbers

[6]http://apkpure.com/cn

[7]For a given activity/fragment class, we determine whether it was visited by checking whether the class was loaded in any `.hprof` file.

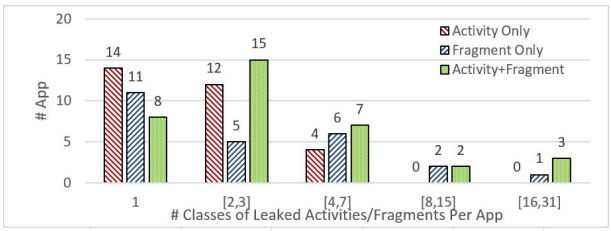


Figure 3. Distribution of leaked classes per App

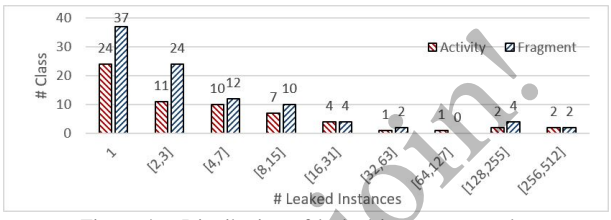


Figure 4. Distribution of leaked instances per class

of leaked activity classes and leaked fragment classes per app were 6 and 16 accordingly. More detailed results are demonstrated in Figure.3. We could see that most of leaking apps leaked less than 8 classes of Activity/Fragment.

Table VII
APPS WITH LEAKED INSTANCES

	Leaked Activity/Fragment					
	Activity (30 apps)		Fragment (25 apps)		Activity+ Fragment (35 apps)	
	#Class	#Instance	#Class	#Instance	#Class	#Instance
Max	6	374	16	926	21	1,075
Min	1	1	1	1	1	1
Avg	2.07	47.37	3.80	67.68	4.49	88.94
Median	2	7	2	4	2	8
Total	62	1,421	95	1,692	157	3,113

Figure.4 shows the distribution of detected instances of each leaked classes. 24 leaked activity classes and 37 leaked fragment classes had exact one instance detected as leaked; Meanwhile, there were totally 61 leaked classes, each of which having at least 4 leaked instances detected. We reported 52 of them to developers of 13 apps accordingly[8]. By the time the paper was submitted, we have received replies from developers of 3 apps confirming all of the reported 11 leaked classes.

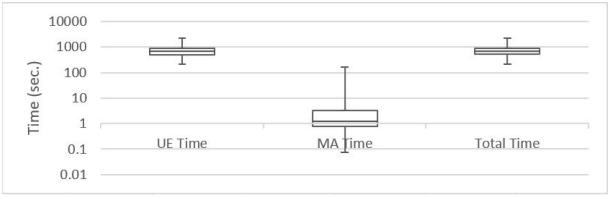
3) Memory Retained: As shown in Table.VIII, generally, memories retained by leaked activities were much more than those retained by fragments. The smallest memory retained by one single leaked activity or fragment in our experiment were no more than 1 KB, while the maximum ones were about 200 KB. Meanwhile, the maximum memories retained by one single class of (leaked) activity or fragment were over 22 MB or 0.5 MB accordingly. Although, apps leaked only about 803 KB on average in our experiments, there was an app "`mazs.studio.linetheme`" having 374 leaked instances of 2 activity classes, which retained more than 22 MB memories in total.

4) Efficiency: Statics of execution time for LeakDAF to run the 64 apps are shown in Figure.5(a). Obviously,

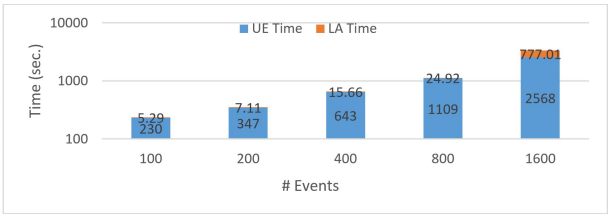
[8]We can not found contact information of the 6 apps containing the remaining 9 classes

Table VIII
RETAINED MEMORY BY LEAKED INSTANCES, CLASSES AND APPS (UNIT:KB)

	Granularity						
	Instance		Class		App		
	Act.	Fgmt.	Act.	Fgmt.	Act.	Fgmt.	Act.+ Fgmt.
Max	220	198	22,004	577	22,015	962	22,015
Min	0.74	0.16	0.95	0.16	3.01	0.16	0.18
Avg	18.39	1.17	421.42	20.80	870.94	79.04	802.98
Median	3.06	0.52	10.86	1.83	48.05	6.14	44.85
Total	26,128	1,976	26,128	1,976	26,128	1,976	28,104



(a) Statics of execution time



(b) Execution time with different numbers of events

Figure 5. Execution time of LeakDAF

the total running time of each app was dominated by the time for UE to execute the app, which required about 734 seconds on average. The most time consuming app took about 2223 seconds (2221 seconds for UE). The time required by LA is affected by the number of `.hprof` files, the number of (leaked) activities/fragments remaining in those files and needing to be checked as well as the numbers and lengths of their paths to GC roots. Take the app "`mazs.studio.linetheme`" for instance, it took LA almost 163 seconds to analyze its 7 dumped `.hprof` files consisting of over 374 leaked instances. However, the average time for LA was only about 6.7 seconds.

We also applied LeakDAF to the app "`com.google.android.apps.genie.geniewidget`" 5 times with different numbers of triggered events and recorded the execution time accordingly. As shown in Figure.5(b), the time required by UE grows linearly with the increase of the number of events. At the same time, the time used by LA keeps low except for the case where #Events = 1600, where 2 `.hprof` files consisting of a great number (over 2000) of leaked activities and fragments were checked. If we don't require all the L-paths, the time can be decreased dramatically by letting the procedure `filterPaths()` return immediately once an L-path is detected.

C. Threats to Validity

Like any empirical study, there are potential threats to the validity of the results of our experiments.

First, as the support libraries containing `android.support.v4.app.Fragment` are packaged within the app's `.apk` file, they can be obfuscated. In such situations, we cannot directly check a fragment's states via the field names. We address the problem by counting the numbers of fields that are probably reset. Briefly, we simply count four numbers $\#_{int(-1)}$, $\#_{int(0)}$, $\#_{bool}$ and $\#_{Obj}$ as shown in Table.IX. If a fragment is successfully destroyed, the following 4 conditions must be true: $\#_{int(-1)} \geq 1$, $\#_{int(0)} \geq 3$, $\#_{bool} \geq 11$ and $\#_{Obj} \geq 6$. We aggressively judge the fragment as destroyed if all the above 4 inequalities hold at the same time. Although it may introduce some false positives, we believe these situations would not happen frequently. Furthermore, if the modified/obfuscated version keeps some names of the fields shown in Table.I but changes their types or usages(e.g., exchange the names of two fields), LA would miss many actually destroyed fragments. We believe such *name-rebinding* modifications seldom happens, and, if simply applying obfuscation tools like Proguard, in most cases LA would not miss any destroyed fragment.

Table IX
EMULATOR CONFIGURATION

Notation	Description
$\#_{int(-1)}$	# of field of type `int` with value equals to -1
$\#_{int(0)}$	# of field of type `int` with value equals to 0
$\#_{boolean}$	# of field of type `boolean` with value equals to `false`;
$\#_{Obj}$	# of field of type `Object` with value equals to `null`;

Second, Dynodroid used in our first experiment is re-implemented by ourselves for running apps requiring Android API newer than provided by original Dynodroid, and the results might not be 100% in consonance with original Dynodroid. However, our version is based on the source code of Dynodroid, only using UiAutomator as well as "dumpsys" command for obtaining candidate UI/system events and executing events; Besides, LeakDAF is also implemented based on the same modified version. Therefore, we believe that the first experiment is fair as well.

Third, like most empirical studies, our results will not necessarily generalise beyond the 35 open source apps and 64 real-world apps to which we have applied LeakDAF.

Fourth, we also only evaluated LeakDAF on a single version of Android platform, and our approach may fail if mechanisms for managing activities/fragments changes in subsequent versions.

D. Noticealbe Findings

- **Intra-app Patterns**: We found many cases, where leaked instances of different classes within one app usually shared the same patterns in their L-paths. Take the app `com.tomatodev.timerdroid` for instance. Each of the three leaked fragment classes(i.e., 6, 7, 8 of Table.V) defined an anonymous inner class, whose instances would be further used as keys by $obj_{App}.mLoadedApk.mServices$ for managing given

bound services. Here obj_{App} indicates the global static `Application` instance of the app. Thus, instances of the 3 fragment classes cannot be freed up. Furthermore, they held references to instances of the remaining 4 classes (i.e., 2,3,4,5 of Table.V), preventing the later from being released.

- **Cross-app Patterns**: Although we've filtered out the paths matching any known patterns, we still found some common patterns shared by leaked instances of different apps. We thought that these patterns should be related to the underlying Android platform, and we plan to find a way to automatically distinguish leaks caused by apps from those caused by the Android platform in our future work.

V. Related Work

A. UI Testing for Android

Tools and like MonkeyRunner[18], Robotium[19], UiAutomator [6], etc., provide capabilities of testing user interface (UI) efficiently by creating automated functional UI test cases that can be run against apps on one or more devices. Monkey[20], a fuzz testing tool included in the Android platform, is a program that runs on an emulator or device and generates *pseudo-random* streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. Dynodroid[5] works in an *observe-select-execute* cycle, in which it first observes the layout of widgets on the current screen and what events are relevant to the app in the current state, then selects one of those events, and finally executes the selected event to yield a new state in which it repeats this process.

Model-based UI testing harnesses human and framework knowledge to abstract input space of an app's GUI, and thus reduces redundancy and improves efficiency[5]. Some Model-based UI testing tools (e.g. [21]) require a model of the app's GUI beforehand; while some others[22] [5] infer GUI models automatically at runtime. [23] proposed a set of multi-level GUI Comparion Criteria providing the selection of multiple abstraction levels for GUI model generation.

A recent work [24] introduced a novel multi-objective search-based testing technique and tool for automated Android app testing.

B. Memory and Resource Leak Detection

There have been many works on detecting memory leaks in Java programs. LeakBot[25] detects memory leaks by formulating structural and temporal properties of reference graphs. [26] [27] try to optimize the static analysis to produce precise leak reports based on some observations about containers/loops. An interesting project, Leak Pruning[28], tries to bound the memory consumption of Java programs with leaks by pruning likely leaked data structures when a program runs out of memory. Relda2[29] provides a light-weight static tool for analyzing potential resource leaks caused by missing release operations for resources. [30] [31] proposed a systematic GUI model-based approach for testing resource leaks for Android apps. It generates comprehensive test cases to trigger repeated executions of neutral cycles (special sequences of GUI events) to discover resource leaks in apps. [32] goes a step further by prioritizing test cases according to their likelihood to cause memory leaks in a given test suite.

General tools for analyzing memory(e.g., MAT[1], AndroMat[7], HAHA [33]) simply report objects or set of objects which are suspiciously big as leak suspects, and do not provide specific supports for detecting leaked activities/fragments of Android; besides, whether the leaks can be detected and how long the process would take depend highly on the experience of the developer/tester.

LeakCanary[2], built on top of HAHA, is a open source memory leak detection library specific for Android. It works as a demo thread and introduces a `RefWatcher` to watch all destroyed activities/fragments. Once an activity/fragment is destroyed, LeakCanary triggers GC and checks whether the activity/fragment remains in memory (by checking whether it is still available from an object of type `KeyedWeakReference` introduced by LeakCanary). If yes, LeakCanary would dump the heap of the app under test, and further analyze the heap dump with HAHA, which locates the activity/fragment object and further determines if it is leaked. However, to use LeakCanary, a developer needs to modify part of the source codes (required code injection/modification might be missed and tedious especially when a great many Fragments exist) and adds given building dependences into the `build.gradle` file of the app; besides, manual interactions are required to run the app.

Compared to the work mentioned, LeakDAF is an automated tool specifically for detecting leaked activities and fragments of Android apps. In a sense, our UiExplorer, enhanced with the *back/re-trigger* and *rotation* mechanisms, can be viewed as an automatic random generator of neutral cycles that destroy and recreate activities and fragments. Unlike LeakCanary working as a library, LeakDAF works as a stand-alone tool and can be directly applied to any app (having GUI) released in markets without requiring the source code.

VI. Conclusion and Future Work

This paper proposed LeakDAF, an automated tool for detecting leaked activities/fragments in an Android app. LeakDAF consists of a UiExplorer which tries to reveal leaks while executing automatically the app under test and a Leak Analyzer that analyzes dumped `.hprof` files for identifying leaked instances. Without requiring source code, LeakDAF not only benefits developers/testers, but also provides app market administrators a simple way for evaluating memory leaks of published apps. By applying it to both open source

and commercial apps, we have shown the effectiveness of LeakDAF.

As UE plays an important role in LeakDAF, we plan to seek more effective and efficient techniques for driving the executions of apps. Besides, we found that a service may leak as well and plan to extend our work to detecting leaked services. Lastly, we plan to find a way to automatically distinguish leaks caused by apps from those caused by the Android platform.

ACKNOWLEDGMENT

This work was supported in part by the National Key Research and Development Plan of China under Grant No. 2016YFB1000802, the National High-Tech Research and Development Program of China under Grant No. 2015AA01A203, the National Natural Science Foundation of China under Grant Nos. 61373011, 61690204, and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

REFERENCES

[1] "Memory analyzer (mat)," http://www.eclipse.org/mat/, accessed July 11, 2016.

[2] "Leakcanary," https://github.com/square/leakcanary, accessed July 11, 2016.

[3] B. R. Preiss, *Data Structures and Algorithms*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[4] "Lock pattern generator," https://f-droid.org/wiki/page/in.shick.lockpatterngenerator, accessed Aug. 9, 2016.

[5] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.

[6] "Uiautomator," https://stuff.mit.edu/afs/sipb/project/android/docs/tools/help/uiautomator/index.html, accessed July 31, 2016.

[7] "Andromat," https://bitbucket.org/joebowbeer/andromat/overview, accessed July 31, 2016.

[8] "Platform versions of android," https://developer.android.com/about/dashboards/index.html, accessed August 8, 2016.

[9] "Myhackerspace," https://f-droid.org/wiki/page/ch.fixme.status, accessed Aug. 9, 2016.

[10] "Timerdroid," https://f-droid.org/wiki/page/com.tomatodev.timerdroid, accessed Aug. 9, 2016.

[11] "Androsense2," https://f-droid.org/wiki/page/com.tritop.androsense2, accessed Aug. 9, 2016.

[12] "List my apps," https://f-droid.org/wiki/page/de.onyxbits.listmyapps, accessed Aug. 9, 2016.

[13] "Opentasks," https://f-droid.org/wiki/page/org.dmfs.tasks, accessed Aug. 9, 2016.

[14] "Micopi+," https://f-droid.org/wiki/page/com.easytarget.micopi, accessed Aug. 9, 2016.

[15] "Alogcat," https://f-droid.org/wiki/page/org.jtb.alogcat, accessed Aug. 9, 2016.

[16] "Anymemo," https://f-droid.org/wiki/page/org.liberty.android.fantastischmemo, accessed Aug. 9, 2016.

[17] "Sage," https://f-droid.org/wiki/page/org.sagemath.droid, accessed Aug. 9, 2016.

[18] "Monkeyrunner," https://stuff.mit.edu/afs/sipb/project/android/docs/tools/help/monkeyrunner_concepts.html, accessed July 31, 2016.

[19] "Robotium," https://github.com/RobotiumTech/robotium, accessed July 31, 2016.

[20] "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey.html, accessed July 31, 2016.

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

[22] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," 2012.

[23] Y.-M. Baek and D.-H. Bae, "Automated model-based android gui testing using multi-level gui comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 238–249. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970313

[24] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proc. of ISSTA'16*, 2016, pp. 94–105.

[25] N. Mitchell and G. Sevitsky, "Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications," in *European Conference on Object-Oriented Programming*. Springer, 2003, pp. 351–377.

[26] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, pp. 151–160.

[27] D. Yan, G. Xu, S. Yang, and A. Rountev, "Leakchecker: Practical static memory leak detection for managed languages," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 87.

[28] M. D. Bond and K. S. McKinley, "Leak pruning," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 277–288. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508277

[29] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, "Light-weight, inter-procedural and callback-aware resource leak detection for android apps," *IEEE Transactions on Software Engineering*, pp. 1–1, 2016.

[30] D. Yan, S. Yang, and A. Rountev, "Systematic testing for resource leaks in android applications," pp. 411–420, 2013.

[31] H. Zhang, H. Wu, and A. Rountev, "Automated test generation for detection of leaks in android applications," pp. 64–70, 2016.

[32] J. Qian and D. Zhou, "Prioritizing test cases for memory leaks in android applications," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 869–882, 2016.

[33] "Headless android heap analyzer(haha)," https://github.com/square/haha, accessed July 31, 2016.