ISSN No. 2456-0472
The Engineering Journal of Application & Scopes, Volume 2, Issue 1, February 2017

ANDROID APPLICATION MEMORY LEAKAGE DETECTION APPROACH

Dhaneshwar Kumar* and Sachin Saxena
Department of Computer Science
Rajshree Institute of Management and Technology, Bareilly, India
* Corresponding author. Tel.: +91-8439516563, E-mail addresses: dhannesh@gmail.com

Abstract—**Android applications run on mobile devices that have limited memory resources. Although Android has its own memory manager with garbage collection support, many applications currently suffer from memory leak vulnerabilities. These applications may crash due to out of memory error while running. Testing of memory leak can detect the vulnerability early. In this paper we perform memory leak testing of Android applications. We first develop some common memory leak patterns specific to Android applications. Then, based on the patterns, we generate test cases to emulate the memory leak. We evaluated the proposed testing approach (denoted as fuzz testing) for a number of Android applications. The initial results indicate that the proposed testing approach can effectively discover memory leaks in applications. Further, implemented code often lacks exception handling mechanism for altered resources and failed invocation of memory management related API calls.**

Keywords- Memory leak, Android, Appication fuzzing, Resource fuzzing, API fuzzing.

I. INTRODUCTION

Android is an open source Operating System (OS) for mobile devices (smart phones) which got popular over the last few years. A recent report suggests that Android occupies more than 50% market share of mobile devices [1]. Android allows users to download and install applications from various sources (e.g., Google Play Store [2]) for game, entertainment, and communication.

Unfortunately, many Android programs suffer from vulnerabilities in the implementation. In particular, memory leak caused by running applications is a subtle vulnerability that often results in unwanted consequences such as application crashes. Memory leak happens when allocated objects in an application live longer than the expected lifetime inside an Activity (an entity for running an application). Keeping the reference of unused memory objects (e.g., drawable images attached to Views) for a longer time results in the denial of new memory allocation requests. Despite Android has built-in memory manager having garbage collection support, it is not adequate to ensure that allocated resources are being managed properly. Much of the burden is on the application developer to understand different types of leaks and avoiding them. This paper is intended to address the detection of memory leak problem based on a testing approach.

In the literature, many testing approaches have been proposed for Android applications [3, 4, 5, 6, 7, 8, 9, 19, 20]. These work focus on activity lifecycle and functionality testing. None of these approaches focus on testing memory leak. Moreover, unlike traditional testing, memory leak mostly has nothing to do with supplying known inputs and then confirming whether the generated output matches with the expected one or not. Further, there has been very little organized discussion in the literature that classifies the common patterns of memory leak and their corresponding fixing approaches.

In this work, we first identify a set of memory leak patterns for Android applications. We show code-level examples of memory leak based on a number of existing bug reports. The code-level patterns enable us to understand the detailed cause of the memory leak including the responsible Android specific objects (e.g., *ImageView*, Bitmap), location of the memory leak (e.g., *onCreate*() of an Activity class), the expected locations where leaks can be prevented (e.g., *onDestroy()* of an Activity class), the role of user actions (e.g., rotating an application frequently), resources (e.g., images stored in resource folder), and API calls used for managing memory objects (e.g., *System.gc()*). Then, we propose to apply the concept of fuzz testing [10] to emulate memory leaks and discover the vulnerabilities. In particular, we propose three types of fuzzing: (i) application fuzzing, (ii) resource fuzzing, and (iii) API fuzzing.

We apply the proposed fuzz testing approach for a set of real-world Android applications. The initial results show that many of the tested applications crash due to memory leak. Further, application developers rarely consider unusual interactions such as launching multiple instances of an application, and rotating an application frequently. Also, most applications do not handle specific exceptions such as failure in allocating memory objects, and checking the status of API calls responsible for releasing memory objects. The paper is organized as follows: Section II discusses an example of memory leak in Android application and the literature work. Section III discusses five types of memory leak patterns and suggested fixing of these leaks. Section IV provides an overview of the fuzz testing approach for revealing memory leak vulnerabilities. Section V shows the evaluation results. Finally, Section VI concludes the paper.

II. BACKGROUND AND RELATED WORK

A. *An example of memory leak in Android application*

Figure 1 shows a code snippet that is vulnerable to memory leak. The *onCreate* method of the activity (*MainActivity*) is invoked when the application starts. One megabyte (1MB) of memory is allocated in a private variable m which is defined inside an event listener object subsequently attached to a button (R.id.button). The onDestroy method is invoked when the activity is destroyed.

```
public class MainActivity extends Activity {
public void onCreate (Bundle bundle) {
super.onCreate (bundle);
setContentView (R.layout.main);
findViewById (R.id.button).setOnClickListener
(new
View.OnClickListener(){
private byte [] m = new byte [1024*1024]; //1
MB is allocated
… …
});
}
public void onDestroy() {
super.onDestroy(); // m still connected to the
listener object
}
}
```

Figure 1. Example of an Android program code having memory leak.

In this implementation, no explicit operation is being done to free up the allocated 1MB memory referenced by m. As a result, the associated activity remains live as it refers to the memory object. The garbage collector is not able to recollect the memory object. The heap or free memory area may run out of available memory. If the activity is launched for a number of times, the application will crash. See Section III.B for fixing this leak.

B. Related work on Android application testing

There are few literature work related to the testing of Android applications. These include conforming application's expected activity lifecycle [4], testing based on event sequence applied to visible GUI objects [3], stress testing of applications by supplying randomly generated events such as clicks, touches, and gestures (Monkey [5]), unit testing of implemented functionalities (Roboelectric [6]), acceptance testing (Robotium [7]), testing of application in the cloud (Testdroid [8]), and generating sequence of events based on the concolic testing concept [9]. These approaches are complementary to our work. In contrast, we focus on testing a non-functional property.

Our work is mostly motivated by fuzz testing (or fuzzing) approach. Fuzzing is a negative testing approach, where a tester identifies the expected input type and format that can be processed by an application in the beginning. The input data type and syntax format are then modified to generate malformed inputs and they are supplied to the application with the hope of discovering bugs.

Some literature works apply the concept of fuzz testing for Android applications [19, 20, 21, 22]. Mahmood et al. [19] perform a white box level fuzzing by exploring call flow graph and supplying primitive data type (number, string) in input taking widgets to discover anomalies in input processing functions. Maji et al. [20] fuzz the IPC communication entities (e.g., intents) to assess the robustness of applications while dealing with incomplete or incorrect intent entities. However, the approach is not suitable for memory leak fuzzing. Vuontisjärvi et al. [21] perform SMS message fuzzing and show that smartphone devices remain vulnerable when they receive malformed messages. ADBFuzz [22] is a fuzzer that generates input randomly and without any specific heuristics or knowledge from a tester. In contrast, we apply the concept of fuzz testing for detecting memory leaks in Android applications.

We are aware of the literature work that apply the concept of fuzz testing to find security bugs and vulnerabilities over the last two decades (interested readers are suggested to see [23]). These include testing of Adobe Reader application by supplying malformed PDF files [11], testing of IDS signature detection capabilities by altering the protocol data unit [12], fuzzing input data to trigger buffer overflow vulnerability [13, 14], fuzzing program execution environment (file system, network) to introduce vulnerability [15], altering internal program variable states during program runtime [16], supplying malformed files based on the syntax knowledge of HTML and Vector image (WMF) files to test the robustness of file processing applications

[17], and altering the PDUs based on syntax to test routing protocol daemons [18]. Despite these work are not relevant to Android application testing, their concepts have influenced our proposed testing approach.

Finally, we are also aware of the Eclipse MAT tool1 that can analyze heap memory dumps. It reports suspected objects consuming high amount of memory. However, the tool is not intended to emulate or generate test cases to replicate memory leak situations.

III. MEMORY LEAK PATTERNS AND FIXING

In this section, we discuss some common causes of memory leaks along with code examples followed by their fixing. We mostly rely on the information from bug reports voluntarily reported by expert Android developers in the website Stack Overflow2. As of writing this article, we find that the website has 369,691 threads discussing various types of bugs related to Android applications. Among them only 83 threads are relevant to memory leak. Being the number relatively small, we manually study these reports to identify the leak patterns. We also explore few other sources [24, 25]. We discuss five types of memory leak patterns and their fixing in Sections III.A – III.E.

A. Memory leak through Bitmap and ImageView (M1)

A Bitmap object represents an image file having pixel color information retrieved from a stored file. A bitmap can be drawn as the background image of a GUI object such as a button. The code snippet of Figure 2 shows an activity (DrawBitmap) that loads a bitmap in the onDraw method of an attached View named BitMapView. In particular, a bitmap image (R.drawable.bitmap) is loaded through the method call BitmapFactory.decodeResource() and saved in the bitMap object. The object is drawn in the default canvas object using the method call drawBitmap with the specified left and top location (1, 1) and with a null Paint object. Though rendered inside the View class, the loaded image (bitMap) remains throughout the lifetime of the activity.

The occupied memory of the bitMap object should be released inside *onDestroy()* method. The *onDestroy()* method of Figure 2 does not have any specific memory release operation. This would start a memory leak if an application is closed and then launched again. The closed Activities and all subsequent Views and the allocated objects still remain in the memory. This will result in the crash of the application due to Out of Memory (OOM) error. If the bitmap size is very large and the invocation of the onDestroy is delayed (e.g., launching a sub-activity from the current activity), the free memory runs out very fast.

```
    public class DrawBitmap extends Activity {
    Bitmap bitMap;
    public void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    setContentView(new BitmapView(this));
    }
    class BitmapView extends View {
    … … …
    public void onDraw(Canvas canvas) {
    Bitmap bitMap =
BitmapFactory.decodeResource
    (getResources(), R.drawable.bitmap);
    canvas.drawBitmap (bitMap, 1, 1, null);
    }
    }
    public void onDestroy() {
    super.onDestroy();
    }
    }
```

Figure 2. Code snippet of drawing a bitmap having memory leak.

One approach of avoiding memory leak is to invoke the recycle() method of an allocated Bitmap object that is not further needed inside the code. This allows freeing up of unused objects even before the invocation of the *onDestroy()* method. An example of recycle() method call is shown in Figure 3. The code can be placed anywhere in the application (View or Activity class).

The second approach to prevent the memory leak is to invoke the garbage collector *System.gc()* immediately before the allocation of a new bitmap object. This method forces the built-in garbage collector to reclaim unused objects at the time of the call. In general, the garbage collector is invoked periodically, and the interval might be too long compared to the rate of memory free and allocation operations being performed by applications. Thus, the most recently freed memory objects may not be reclaimed right away. Figure 4 shows an example call of garbage collector before creating the bitmap based on Figure 2 example. Similar to the BitMap object, an application can load an image as an *ImageView* object during runtime and add them to a layout (inside View class). Figure 5 shows two examples of loading image as *ImageView* object. The first line displays an image by using built-in image (R.id.image) view that takes care of the loading based on the current layout of an application. The R.id.image includes the description of the image source image and specific properties (e.g., width, height, position) that are used during the rendering process. The second line loads an image file named test.png from a resource folder (res/drawable).

Similar to the BitMap object, an allocated *ImageView* object needs to be released through the recycle() method call in *onDestroy()* method, and garbage collection (*System.gc()*) call should be invoked before allocating new resources.

```
    if (bitMap != null) {
    bitMap.recycle();
    bitMap = null;
    }
```

Figure 3. Example of recycle method call.

```
    public void onDraw(Canvas canvas) {
    System.gc();
    Bitmap bitMap =
BitmapFactory.decodeResource
    (getResources(), R.drawable.bitmap);
    }
```

Figure 4. Example of garbage collector method invocation.

```
    ImageView imgView = (ImageView)
findViewById (R.id.image);
    imgView.setImageResource (R.drawable.test);
```

Figure 5. Code example of loading images as *ImageView* objects.

B. Memory leak through event listener (M2)

Memory can be allocated inside an event listener object. If the allocated memory is not freed via setting the null value to the listener object, the Activity or View class that is having the listener object cannot be destroyed, and subsequently remains in the memory causing memory leak.

```
    public class Main extends Activity {
    public void onCreate(Bundle bundle) {
    setContentView(R.layout.main);
    findViewById(R.id.button).setOnClickListener (
    new View.OnClickListener() {
    private byte[] m = new byte[1024*1024]; //1 MB
allocated
    public void onClick(View v) { … }
    });
    }
    }
```

Figure 6. An example of memory leak via event listener object.

```
    public void onDestroy() {
    findViewById(R.id.button).setOnClickListener
(null);
    super.onDestroy();
    }
```

Figure 7. An example of fixing memory leak for event handler object.

We show a code snippet of memory leak inside an event listener object in Figure 6. Here, the Main activity attaches a click event listener object (setOnClickListener() method call3) for a button (R.id.button). Inside the event listener, one megabyte of memory is allocated (the m variable). To avoid the memory leak, the onDestroy method should include the necessary code to set the listener object as null. We show an example of fixing the memory leak in Figure 7 via setOnClickListener(null) method call.

C. Memory leak through animation activity (M3)

A Drawable object is used to perform animation activity (e.g., drawing on the screen). It can also be used to retrieve resources based on a set of APIs. Figure 8 shows an 3 example of a Drawable object (d) creation inside the *onCreate*() method, which subsequently attaches the background of a TextView object (t).

```
    public void onCreate (Bundle state) {
    super.onCreate(state);
    TextView t = new TextView(this);
    Drawable d = getDrawable(R.drawable.img1);
    t.setBackgroundDrawable(d);
    }
```

Figure 8. An example of creating a drawable object.

A Drawable object requires that the attached View implements a call back interface to perform the animation task. As a result, a reference of the call back interface for a Drawable object remains in memory even though an Activity is destroyed. Thus, the subsequent View and Activity classes remain in the memory, and unfortunately the garbage collector is not able to reclaim these objects. To fix this type of memory leak, the onDestroy method should enumerate all drawable objects and then set the null value for all the relevant callback objects. This technique is known as unbinding of the callback. Figure 9 (a) shows an example code for unbinding the call back method of one Drawable object d (based on the example of Figure 8).

Similarly, *setBackgroundDrawable()* is another method (used for animation in the background) that also needs to be un-binded explicitly inside the *onDestroy()* method. Figure 9(b) shows an example code of releasing all callback objects from View class (view) via setBackgroundDrawable(null).

public void
onDestroy(){
super.*onDestroy()*;
if (d != null)
d.setCallback(null);
}
	public void *onDestroy()*{
	super.*onDestroy()*;
	if(view != null{
	ViewGroup viewGroup =
	(ViewGroup)view;
	int childCount =
	viewGroup.getChildCount();
	for(int index = 0; index <
	childCount;
	index++){
	View child =
	viewGroup.getChildAt(index);
	child.setBackgroundDrawable(
	null);
	}
	}

(a) Fixing memory leak for one object	**(b) Fixing memory leak for multiple objects**

Figure 9. Example code of fixing memory leak due to drawable object.

D. Memory leak due to static object (M4)

A static variable is intended to remain throughout the lifetime of an application. Unfortunately, it is not garbage collected right away after an application is terminated.

```
public class Main extends Activity{
static Bitmap bmp;
public void onCreate (){
bmp = BitmapFactory.decodeResource
(getResources(),
R.drawable.bitmap);
…
}
public void onDestroy(){
super.onDestroy();
bmp = null;
}
}
```

Figure 10. Example of memory leak due to a static memory object.

```
public void onDestroy(){
in.bmp = null;
in = null;
System.exit (0);
}
```

Figure 11. Example of memory leak fixing for a static object.

Figure 10 shows a code example where the Main activity defines a static Bitmap object variable (bmp). The *onCreate*() method loads a bitmap object in bmp. The *onDestroy()* method destroys the allocated bitmap object by nullifying it. However, being a static object, it would still remain in the memory and result in memory leak.

This type of leak can be avoided by invoking the *System.exit()* method to reclaim the freed memory right away. The *onDestroy()* method should include the call. Figure 11 shows the revised *onDestroy()* method to avoid the memory leak due to static object allocation. Note that a memory object declared as static and private inside an inner class will always be leaked. The best approach to avoid memory leak due to private static object type is to avoid defining them at all if possible. If static variables need to be used, then the static objects should not occupy large chunk of memory. Similar discussion applies for constant objects.

E. Memory leak through AdView object (M5)

AdView is a special view object intended to display advertisements and enable a user to interact with the advertisements based on touch. Figure 12 shows a code example for creating an *AdView* object (adv) inside the *onCreate*() method. Here, the advertisement is loaded via the *loadAd()* method call. The object is then added in the main layout (R.id.mainLayout) of the View. The *onDestroy()* method is the place where the allocated *AdView* object needs to be freed. In the example, the allocated object is not being freed explicitly. Thus, if the application is rotated, the current activity gets destroyed and starts a new activity. However, the allocated object for the *AdView* still remains in the memory. This results in a memory leak. To fix this memory leak, the destroy method has to be called explicitly followed by setting the object as null. Figure 13 shows the *onDestroy()* method that prevents the leak.

Table I summarizes the five memory leak patterns (M1- M5) based on three attributes (columns 2-4): Android objects for the memory leak, memory allocation location, and the expected location of memory release. The last three columns show the relevancy of three types of fuzz testing (application, resource, and API) that might reveal memory leak. In Section 4, we discuss these fuzz testing types.

```
public class Main extends Activity {
public AdView AdView;
public void onCreate(Bundle bundle) {
adv = new AdView(this, AdSize.BANNER,
PUBLISHER_ID);
adv.loadAd(new AdRequest()); //loading
advertisement
(LinearLayout)findViewById(R.id.mainLayout).add
View(adv);
}
public void onDestroy() {
super.onDestroy();
}
}
```

Figure 12. Example of memory leak through *AdView* object.

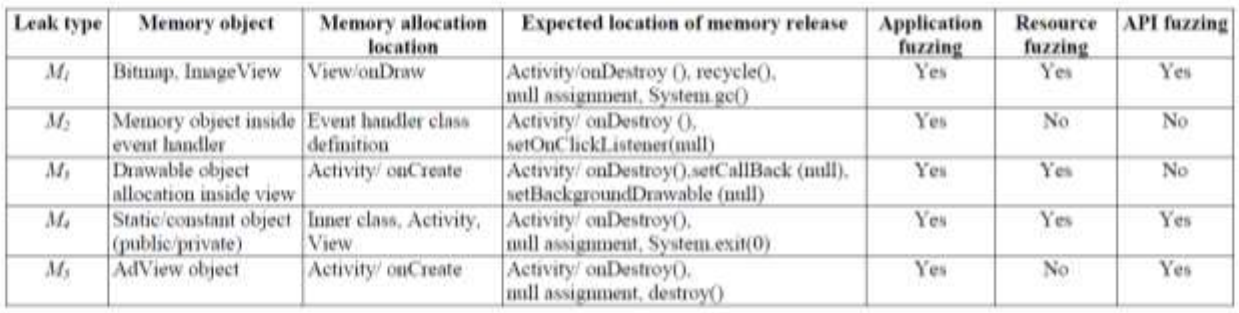
```
public void onDestroy() {
if (adv != null){
adv.destroy();
adv = null;
}
super.onDestroy();
}
```

Figure 13. Example of memory leak fixing of the *AdView* object.

IV. MEMORY LEAK PATTERN GUIDED FUZZ TESTING

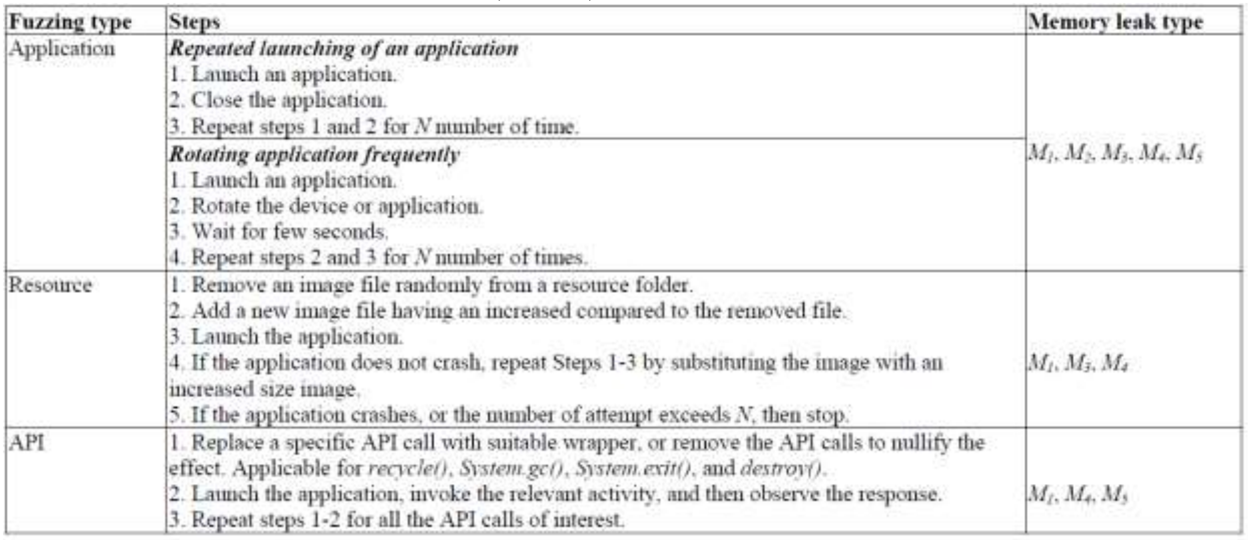
In this section, we rely on the common memory leak patterns to emulate memory leak in running applications. We choose fuzz testing to discover memory leak. This technique is suitable due to our observation that memory leak is not strictly related with supplying specific malformed inputs and conforming expected output results.

TABLE I. MEMORY LEAK PATTERNS AND FUZZ TESTING TYPES



Leak type	Memory object	Memory allocation location	Expected location of memory release	Application fuzzing	Resource fuzzing	API fuzzing
M_1	Bitmap, ImageView	View/onDraw	Activity/onDestroy (), recycle(), null assignment, System.gc()	Yes	Yes	Yes
M_2	Memory object inside event handler	Event handler class definition	Activity/ onDestroy (), setOnClickListener(null)	Yes	No	No
M_3	Drawable object allocation inside view	Activity/ onCreate	Activity/ onDestroy(),setCallBack (null), setBackgroundDrawable (null)	Yes	Yes	No
M_4	Static/constant object (public/private)	Inner class, Activity, View	Activity/ onDestroy(), null assignment, System.exit(0)	Yes	Yes	Yes
M_5	AdView object	Activity/ onCreate	Activity/ onDestroy(), null assignment, destroy()	Yes	No	Yes

TABLE II. FUZZ TESTING TYPES, STEPS, AND DETECTED MEMORY LEAK TYPES



Fuzzing type	Steps	Memory leak type
Application	***Repeated launching of an application*** 1. Launch an application. 2. Close the application. 3. Repeat steps 1 and 2 for *N* number of time. ***Rotating application frequently*** 1. Launch an application. 2. Rotate the device or application. 3. Wait for few seconds. 4. Repeat steps 2 and 3 for *N* number of times.	M_1, M_2, M_3, M_4, M_5
Resource	1. Remove an image file randomly from a resource folder. 2. Add a new image file having an increased compared to the removed file. 3. Launch the application. 4. If the application does not crash, repeat Steps 1-3 by substituting the image with an increased size image. 5. If the application crashes, or the number of attempt exceeds *N*, then stop.	M_1, M_3, M_4
API	1. Replace a specific API call with suitable wrapper, or remove the API calls to nullify the effect. Applicable for *recycle()*, *System.gc()*, *System.exit()*, and *destroy()*. 2. Launch the application, invoke the relevant activity, and then observe the response. 3. Repeat steps 1-2 for all the API calls of interest.	M_1, M_4, M_5

```
public class AppFuzzTest1 extends TestApp1
<MainActivity> {
public void setUp() throws Exception {
int N = 500;
super.setUp();
for (int i=0; i < N; i++){
MainActivity mActivity = getActivity();
mActivity.finish();
}
}
}
```

Figure 14. An example of test case for application fuzzing (repeated launching of an application).

Rather, it is mostly due to the abnormal user level activities which may include destructing activity voluntarily/involuntarily, the wrongful assumption of programmers about built-in garbage collector and lack of understanding on the role of APIs to avoid memory leak, and the wrongful assumption that the application resources will never be altered (e.g., images present in res/drawable folder). We propose to have three types of fuzz testing to emulate memory leak situations: application fuzzing, resource fuzzing, and API fuzzing.

Table II shows the detailed steps to generate test cases for each of the fuzzing types along with the memory leak pattern types detected. The application fuzzing is useful for detecting all the five patterns of leak (M1-M5). The resource fuzzing can detect M1, M3, and M4 types of memory leak. The API fuzzing can be used to detect M1, M4, and M5 types of memory leak. We discuss more on the three types of fuzzing in Sections IV.A-IV.C.

A. Application fuzzing

This test is intended to fuzz user interaction with an application. We apply two separate testing: (i) repeated launching of the same application, and (ii) rotating an active application frequently.

The second column of Table II shows the detailed steps that we define to generate test cases. Note that N is assumed to be a large number (e.g., 500). We rely on the Android testing framework 4 to develop test cases for performing application level fuzzing. JUnit is intended for testing functionalities. However, we are using this framework to test a non-functional property (the presence of memory leak). The Android JUnit extensions allow developing test template to create Activity, invoke actions, and supply inputs. However, for testing of memory leak, we identify that launching, finishing, delaying, rotating activity are some of the basic tasks that need to be implemented to emulate the memory leak.

We show an example of test case for application fuzzing to emulate memory leak by repeatedly launching an application in Figure 14. Here, the setUp() method launches an application (*getActivity*()) and then destroys (mActivity.finish()) the application for 500 times. Here, TestApp1 is the subject application package under test.

B. Resource fuzzing

To perform the resource fuzzing, we decompile application using apktool5, which provides us the original directory structure and sources of the application. We then replace randomly an image with a very large sized image from the res/drawable folder, compile the decompressed application using Eclipse SDK, run the application in the emulator, and observe any exception due to OOM error. The detailed steps of the resource fuzzing are shown in the second row of Table II. This technique requires having an initial collection of images of very large size (e.g., order of 10MB) to force the memory leak and discover the lack of exception handling inside applications due to the incorrect assumption that resources cannot be altered and will be always small in size. Note that the substituted image file has the same name with the newly replaced file.

C. API fuzzing

The detailed steps of the API fuzzing are shown in the third row of Table II. Currently, we only consider four APIs: recycle(), *System.gc()*, System.exit(0), and destroy(). To nullify the effect of the actual function call (i.e., a memory object is not freed despite a method call), wrappers can be implemented and the original method call in the source needs to be replaced with the wrapper. The other option is to simply remove the method calls from the decompiled sources and then compile the applications back to APK packages (similar to the steps for resource fuzzing). We follow the later approach to understand the behavior of an application if APIs related to memory release operations fail. Note that for all three types of testing, we rely on the log information logcat tool to detect the memory leak via Out Of Memory (OOM) error.

V. EVALUATION RESULTS

We evaluated the proposed fuzz testing by randomly collecting 10 Android applications from Google Play Store [2]. The applications are tested by using a device emulator. Some configuration parameters are as follows: platform version 1.6, minimum of API level 4, default screen HVGA, RAM size 64 MB, support of touch screen and camera. Table III shows the characteristics of these applications which include the name, version number, number of java classes, total number of drawable folder, images inside the drawable folder, activity and view.

Table IV shows the characteristics of the applications for M1 memory leak pattern. The second and third columns show the number of Bitmap and *ImageView* objects defined in the applications. The fourth and fifth columns indicate the invocation of decodeResource() and setImageResource() method calls, respectively. The last two columns show the number of recycle() and garbage collector (*System.gc()*) calls, respectively. From the data, we find that most of these applications use *ImageView* as the preferred method of loading image objects using the setImageResource() call.

TABLE III. CHARACTERISTICS OF ANDROID APPLICATIONS

Application	Version	class #	Drawable	Image	Activity	View
Barcode Scanner	4.3.1	311	2	12	5	1
FxCamera	2.5.5	1,837	7	281	7	6
Huffington Post	11.4.0	800	6	432	15	57
My Currency – Converter	3.1.3	352	9	801	9	15
Skype	3.2.0.6673	978	21	3,119	7	25
To-Do Calendar Planner	4.0	1,301	5	420	58	34
Viber	3.1.0.1103	1,489	9	1,030	15	51
Virtual Table Tennis 3D	2.7.3	323	3	121	7	2
WhatsApp	2.10.768	4,134	10	1,781	36	64
YouTube	4.5.17	8,064	17	1,165	30	56

TABLE IV. CHARACTERISTICS RELATED TO M1 LEAK PATTERN

Application	Bitmap	ImageView	decodeResource()	setImageResource()	recycle()	System.gc()
Barcode Scanner	0	0	1	0	1	0
FxCamera	1	2	3	30	22	34
Huffington Post	2	8	0	25	103	3
My Currency Converter	0	2	0	2	24	0
Skype	0	4	1	77	57	0
To-Do Calendar Planner	2	18	0	489	857	0
Viber	9	3	23	21	752	4
Virtual Table Tennis 3D	4	9	0	1	3	4
WhatsApp	0	18	20	144	1,842	2
YouTube	0	8	8	78	1,248	6

Table V shows the characteristics related to M2 and M3leak patterns. The first and second columns show the number of event listener object created and the number of times listener objects are set to null, respectively. The third, fourth, and fifth columns show the number of drawable objects set in the background, the number of times setCallback(null) and setBackgroundDrawable(null) are invoked, respectively.

TABLE V. CHARACTERISTICS FOR M2 AND M3 LEAK PATTERNS

Application	setOnClickListener New	setOnClickListener Null	setBackgroundDrawableR	setCallback (null)	setBackgroundDrawable (null)
Barcode Scanner	1	1	0	0	0
FxCamera	43	1	0	0	0
Huffington Post	53	2	0	1	15
My Currency – Converter	5	0	4	0	0
Skype	84	19	3	2	11
To-Do Calendar Planner	320	0	0	0	1
Viber	65	2	1	3	9
Virtual Table Tennis 3D	21	0	0	0	0
WhatsApp	298	20	2	2	30
YouTube	56	4	0	4	18

TABLE VI. CHARACTERISTICS FOR M4 AND M5 LEAK PATTERNS

Application	publicStatic	privateStatic	systemExit	AdView	destroy
Barcode Scanner	52	319	0	0	0
FxCamera	1,234	811	0	193	11
Huffington Post	5,558	790	1	397	61
My Currency – Converter	1,826	233	0	77	32
Skype	13,650	827	3	0	20
To-Do Calendar Planner	6,747	975	0	0	11
Viber	8,973	2,717	1	0	55
Virtual Table Tennis 3D	619	515	0	431	18
WhatsApp	8,400	2,968	0	0	22
YouTube	7,042	2,400	4	0	0

Table VI shows the characteristics of the tested applications that are relevant to M4 and M5 leak patterns. The second and third columns show the number of public and private static variables defined in these applications. The fourth column shows the number of System.exit(0) method call present in these applications. The last two columns of Table VI show the number of *AdView* object creation and the presence of destroy() method call, respectively.

Table VII shows the summary of our evaluation of the application level fuzz testing. We find the 6 out of 10 applications crash during repeated launching of applications, and 3 crashed due to continuous rotation.

Table VIII shows a summary of the resource fuzzing. Here, we replace a randomly selected image of an application that is loaded at the beginning with a large sized image (10MB, 50MB, and 100MB). To our surprise, we find that as the replaced image size increases, more applications crash. In particular, when the image size exceeds the RAM size (64MB), applications always crash. The implemented code does not handle the exceptional circumstance where the loaded image size can be large and resource allocation fails.

TABLE VII. RESULTS OF APPLICATION FUZZING

Application	Repeated launching	Rotating
Barcode Scanner	Crash	No crash
FxCamera	No crash	No crash
Huffington Post	No crash	No crash
My Currency – Converter	Crash	Crash
Skype	Crash	No crash
To-Do Calendar Planner	Crash	Crash
Viber	No crash	No crash
Virtual Table Tennis 3D	Crash	Crash
WhatsApp	Crash	No crash
YouTube	No crash	No crash

TABLE VIII. RESULTS OF RESOURCE FUZZING

Application	10MB	50MB	100MB
Barcode Scanner	No crash	Crash	Crash
FxCamera	Crash	Crash	Crash
Huffington Post	No crash	No crash	Crash
My Currency – Converter	No crash	Crash	Crash
Skype	No crash	Crash	Crash
To-Do Calendar Planner	Crash	No crash	Crash
Viber	No crash	No crash	Crash
Virtual Table Tennis 3D	Crash	Crash	Crash
WhatsApp	No crash	Crash	Crash
YouTube	Crash	Crash	Crash

Table IX shows a summary of the API fuzzing. We find that these applications rarely handle exceptions for API call invocation failure. The removal of recycle() method results in the crash of all the applications as all of them rely on this method to free up resources from *ImageView* and Bitmap objects. Removing the destroy() method call has the disastrous effect on applications that create *AdView* objects. These applications crash once advertisements are loaded and the applications are destroyed or multiple activities are invoked. Failure to invoke the *System.gc()* and System.exit(0) method calls result in crashes on applications having significant memory allocated through both public and private static variables (e.g., Huffington Post).

TABLE IX. RESULTS OF API FUZZING

Application	recycle ()	System.gc ()	System.exit (0)	destroy ()
Barcode Scanner	Crash	No crash	No crash	No crash
FxCamera	Crash	Crash	No crash	Crash
Huffington Post	Crash	Crash	Crash	Crash
My Currency – Converter	Crash	No crash	Crash	Crash
Skype	Crash	No crash	Crash	No crash
To-Do Calendar Planner	Crash	No crash	Crash	No crash
Viber	Crash	Crash	No crash	No crash
Virtual Table Tennis 3D	Crash	Crash	No crash	Crash
WhatsApp	Crash	Crash	Crash	No crash
YouTube	Crash	Crash	Crash	No crash

VI. CONCLUSIONS AND FUTURE WORK

Android applications are widely used now-a-days. However, most of the applications have subtle memory leak issue which may cause applications to crash. Testing of memory leak is challenging as the underlying causes vary due to various types of objects, resource, and APIs that might be responsible for the leak. Unfortunately, no literature work has addresses a systematic way of testing memory leaks before deploying Android application. This work proposes a fuzz testing method based on identifying five common memory leak patterns: (i) memory leaks via allocated Bitmap or *ImageView* objects, (ii) memory allocation inside event listener object, (iii) callback objects due to drawable animation objects, (iv) static objects, and (v) objects displaying advertisements. We identify common patterns where memory leak can be understood in terms of the responsible Android specific objects, location of memory allocation, and expected location of memory release. The patterns are used to guide the generation test cases to discover memory leaks through application, resource, and API fuzzing. We implement the test cases by leveraging JUnit testing framework of Android. We apply our testing approach for 10 open source Android applications. The results indicate that the approach can effectively discover memory leak vulnerabilities in Android applications. Our initial results identify the memory leak vulnerabilities in the tested applications where developers assume that their resources will not be altered and memory management API invocations are always successfully invoked during application runtime.

Our future work includes developing more memory leak patterns such as context, inner class, and database connection leaks. We plan to convert the leak patterns into test cases automatically to allow the detection of memory leak vulnerabilities early. As part of resource fuzzing, our goal is to alter other resource file types such as video and text files. For API fuzzing, our future goal is to implement wrappers for memory release related method calls for returning anomalous values and test application's behavior. We also plan to test more Android applications for different types of device emulators in the future.

REFERENCES

[1] D. Aaron, "Google Android Passes 50% of Smartphone Sales, Gartner says," http://www.businessweek.com/news/2011-11-17/google2android-passes-50-of-smartphone-sales-gartner-says.html
[2] Google Play Store, https://play.google.com/store
[3] D. Amalfitano, A. Fasolino, and P. Tramontana, "A GUI Crawlingbased Technique for Android Mobile Application Testing," Proc. Of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011, pp. 252-261.
[4] D. Franke, C. Elsemann, S. Kowalewski, and C. Weise, "Reverse Engineering of Mobile Application Lifecycles," Proc. of the 18th IEEE Working Conference on Reverse Engineering (WCRE), 2011, pp. 283-292.
[5] UI/Application Exerciser Monkey, http://developer.android.com/tools/help/monkey.html
[6] Roboelectric: Unit Test of Android Application, http://robolectric.org
[7] Robotium, http://code.google.com/p/robotium/
[8] Testdroid, Automated Testing Tool for Android, http://testdroid.com/
[9] S. Anand, M. Naik, M. Harrold, and H. Yang, "Automated Concolic Testing of Smartphone Apps," Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Nov 2012, pp. 1-11.
[10] A. Takanen, J. Demott, and C. Miller, Fuzzing for Software Security Testing and Quality Assurance, Artech House Tnformation Security and Privacy.
[11] A. Jorgensen, "Testing with Hostile Data Streams," ACM SIGSOFT Software Engineering Notes, Volume 28, Issue 2, March 2003, pp. 9.
[12] G. Vigna, W. Robertson, D. Balzarotti, "Testing Network-based Intrusion Detection Signature Using Mutant Exploits," Proceedings of the ACM Conference on Computer and Communication Security (CCS), October 2004, Washington DC, pp. 21-30.
[13] A. Lanzi, L. Martignoni, M. Monga, and R. Paleari, "A Smart Fuzzer for x86 Executables," Proc. of the 3rd International Workshop on Software Engineering for Secure Systems (SESS'07), Minneapolis, USA, May 2007, pp 7.
[14] X. Zhang, L. Shao, and J. Zheng, "A Novel Method of Software Vulnerability Detection based on Fuzzing Technique," Proc. of the International Conference on Apperceiving Computing and Intelligence Analysis, December 2008, pp. 270-273.
[15] W. Du and A. Mathur, "Testing for Software Vulnerabilities Using Environment Perturbation," Proc. of the International conference on Dependable Systems and Networks (DSN), New York, NY, June 2000, pp. 603-612.
[16] A. Ghosh, T. O'Connor, and G. McGraw, "An Automated Approach for Identifying Potential Vulnerabilities in Software," Proc. of IEEE Symposium on Security and Privacy, Los Alamitos, CA, USA, 1998, pp. 104-14.
[17] H. Kim, Y. Choi, D. Lee, and D. Lee, "Practical Security Testing using File Fuzzing," Proc. of International Conference on Advanced Computing Technologies (ICACT), Hyderabad, India, February 2008, pp. 1304-1307.
[18] O. Tal, S. Knight, and T. Dean, "Syntax-based Vulnerabilities Testing of Frame-based Network Protocols," Proc. of the 2nd Annual Conference on Privacy, Security and Trust, Fredericton, Canada, October 2004, pp. 155-160.
[19] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud," Proc. of the 7th International Workshop on Automation of Software Test (AST), Zurich, Switzerland, June 2012.
[20] A. Maji, F. Arshad, S. Bagchi, and J. Rellermeyer, "An Empirical Study of the Robustness of Inter-component Communication in Android," Proc. of the 42nd Annual IEEE International Conference on Dependable Systems and Networks (DSN), June 2012, pp. 1-12.
[21] M. Vuontisjärvi and T. Rontti, SMS Fuzzing, Accessed from http://www.codenomicon.com/resources/whitepapers/codenomicon_ wp_SMS_fuzzing_02_08_2011.pdf
[22] ADBFuzz – A Fuzz Testing Harness for Firefox Mobile, http://blog.mozilla.org/security/2012/03/09/adbfuzz-a-fuzz-testingharness- for-firefox-mobile/
[23] H. Shahriar and M. Zulkernine, "Mitigation of Program Security Vulnerabilities:

Approaches and Challenges," ACM Computing Surveys (CSUR), Vol. 44, No. 3, Article 11, pp. 1-46, May 2012.

[24] Android Memory Leaks OR Different Ways to Leak, http://blog.evendanan.net/2013/02/Android-Memory-Leaks-ORDifferent- Ways-to-Leak, Feb 2013.

[25] Avoiding memory leaks, Accessed from http://androiddevelopers. blogspot.com/2009/01/avoiding-memory-leaks.html, Jan 2009.