Static Analysis of Memory Leak in Android Applications

Di Zhou [1, a], Zhengyu Fu[2, b]

[1]College of Computer Science and Technology , Nanjing University of Aeronautics and Astronautics, Nanjing, 210016, China

[2] College of Computer Science and Engineering , Dalian Nationality University, Dalian,116650,China

[a]email: dzhou0216@163.com, [b]email:joshuazark@163.com

Keywords: Memory Leak; Static Analysis; Android Applications

Abstract. The popularity of Android applications have grown dramatically in the last few years. 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, and this will lead a bad user experience. The target of our work is static object reference analysis, then we can roughly find whether an application has memory leak bugs, explain the reason of a memory leak problem. After that, we can discovery the location of leak bugs and solve them finally.

Introduction

Android is an open source Operating System for smart mobile phones which became popular over this years. Because of embedded device, usage of Android devices resources is limited just like other embedded systems, especially memory resources. Memory leak caused by running Android applications is a typical vulnerability that often lead to unwanted application crashes.

Memory leak bugs often happen when the objects allocated by an application live longer than the lifetime inside an Activity(an instance for running an application, the important component of Android system). Keeping the reference of unused memory objects(e.g., images attached to Views)for a long time also leads to a memory leak. Despite there is a memory manager for garbage collection in Android, it is not enough to ensure whether allocated resources could be managed correctly.

In this paper, we analysis Android application code to find potential memory leak in the application by static analysis method. We show some examples of memory leak code including the reasons leading to memory cash, location of the memory leak(e.g., onCreate() method call of Activity), the expected locations where leaks can be prevented(e.g., onDestroy() method call of Activity). This paper will help Android application developers find the potential memory and fix them.

Memory Leak Classification

As introduced in the last part, memory leak risky modules can be classified following types.

Java heap memory. Android application is developed by java code, it could have memory leak because of java heap memory. Java heap memory is a place used to store allocated objects. Java has its own memory manager to garbage collect automatically, thus leak happens only when unused objects are referenced unnecessarily. Another resource leaks(e.g., leaking of database objects) also appear in this memory place.

API memory. This memory is used to store native code, which is accessible to Android application coed by API(Application Programming Interface)calls. When developers develop a project ,they are required explicit memory management. For example, prevent leaking of native bitmap objects can call explicitly the native recycle method of the Bitmap class. This kind of memory space is particularly effective to check an many Android devices make large amount of use of code and thus native memory.

Communication component. Android provide some efficient components to communicate with inter-process, such as binder and intent. Actually ,a binder is directly support by the underlying Linux kernel in the Android operating system, which is the core component of a high-performance remote procedure call(RPC)mechanism. When using a binder to communicate needs to require creation of global API references, and these references can be collected by the garbage collector. Unnecessarily these references keeping could lead to potentially large objects. The global API references are destroyed by calling the finalizer of android.os.Binder, so the number of Binder instances is an important indicator to judge whether redundant API references are alive.

Threads. In a GUI(graphical user interface) application, threads are usually created to perform time-consuming operation, in order to keep good reactivity. For example, the e-book reader Vudroid, when open a PDF file, creates a new thread to calculate the data of requested file. A buggy operation could hang thread execution, while new threads are being created. Persistent growth of active threads in an application is an instruction to detect software defects, thus proposed analysis method to collect the number of active threads measurement.

All the discussion of measurement can be easily got through the service provided by the Android system platform, and does not need to modify any application codes or system codes. We can use static analysis method to discover potential memory leak bugs. Next section, we will give some examples of the Android application codes which contain memory leak typically.

Static Analysis Memory Leak Code

In this section, we discuss some common reasons of memory leaks along with code examples by static analysis method. All the examples are from the website Stack Overflow 2

Memory leak through Bitmap and Image View. A bitmap object represents an image file with pixel color information retrieved from a stored file. A bitmap is often used in GUI software, such as a background image for a button.

```
public class BitmapTest extends Activity{
   Btimap bitmap;
   public void onCreate(Bundle bundle){
      super.onCreate(bundle);
      setContentView(new Bitmap View(this));
   }
   class Bitmap View extends View{
      … … …
      public void onDraw(Canvas canvas) {
         Bitmap bitmap=BitmapFactory.decodeResource(getResources(),R.darwable.bitmap);
         canvas.drawBitmap(bitmap,1,1,null);
      }
   }
   public void onDestory(){
      super.onDestory();
   }
 }
```

Figure 1 . Code snippet of drawing a bitmap with memory leak

The code in Figure 1 demonstrates an activity(BitmapTest) that loads a bitmap resource via the onDraw method of an attached Bitmap View .In particular, bitmap image(R.darwable.bitmap)is loaded via the BitmapFactory.decodeResource method call and saved in the bitmap reference object. Through the drawBitmap method call with the specified left and top location(1,1) and with a null Paint object, the bitmap object is drawn in the default canvas object. Although the object presents in the View class, the loading image remains in the whole lifetime of the activity.

The allocated bitmap memory could be released through onDestroy() method. In Figure 1 ,the onDestroy() method does not have any specific memory destroy operation, which would lead to a memory leak when executes this operation repeatedly. When close the activity, the bitmap still keeps alive in the memory. This will lead to a crash of the Android application because of OOM(out of memory)error. If the activity load a very large size bitmap and the onDestroy method call does not execute in time (e.g., user receives a phone call), the application crashes very fast.

One method to avoid memory leak is to call recycle() method, which could allocate the bitmap object that is not useful in the code. This allowing freeing up of unused objects before call the onDestory() method. Figure 2 shows an example of call for recycle() method. This code could be anywhere in the application before call the onDestroy() method.

```
If(bitmap!=null){
   bitmap.recycle();
   bitmap=null;
}
```

Figure 2. Code snippet of call recycle method

Memory leak through communication component. Communication component could lead to memory leak such as event listener object. If the event listener object allocated does not set null instead of destroyed, the Activity or View class still have the object and could not be destroyed, which could subsequently cause a memory leak.

```
public class EventListener extends Activity{
   public void onCreate(Bundle bundle){
      setContentView(R.layout.EventListener);
      findViewById(R.id.button).setOnClickListener(new View.OnClicklistener(){
         private byte[] memory=new byte[512*512];
         public void onClick(View v){…}
   });
   }
}
```

Figure 3 . Code snippet of event listener object with memory leak

The code of Figure 3 demonstrates an memory leak caused by event listener object. The EventListener activity registers an event listener object for a button(R.id.button). Inside the code, a quarter megabyte of memory is allocated to the memory variable. To avoid the memory leak, inside the call for onDestroy method, the listener object must set as null. Figure 4 shows an example to avoid this kind of memory leak.

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

Figure 4 . Code snippet of fixing memory leak for event listener object

Memory leak through static objects. When developers develop an application ,some static objects are defined in the code. Because of static objects remain in the whole lifetime of the application and could not be collected by garbage collector right away, it easily leads to a memory crash.

```
public class StaticMemory extends Activity{
   static Bitmap bitmap;
   public void onCreate(Bundle bundle){
      super.onCreate(bundle);
      bitmap=BitmapFactory.decodeResource(getResources(), R.drawable.bitmap);
      … … …
   }
     public void onDestroy(){
        Super.onDestroy();
        Bmp=null;
   }
 }
```

Figure 5 . Code snippet of static object with memory leak

The code of Figure 5 diplays an memory leak caused by defining a static Bitmap object variable. When the activity launched ,the application loads a bitmap object. The onDestroy method destroys the object by setting it as null. However, the static object would remain in the memory still, thus this will result in memory crash. This kind of leak can be solved via invoking the System.exit() method to collect the allocated memory immediately. Figure 6 shows the solution to avoid memory crash.

```
public void onDestory(){
  bitmap=null;
  System.exit(0);
}
```

Figure 6 . Code snippet of fixing memory leak for static object

Conclusion

Android applications are widely used in now-a-days. Memory leak could lead to the application crash ,which brings a bad user experience. In this paper, we use static analysis method to analysis Android application code ,then find potential memory leak in the application. We show some code-level examples of memory leak including the reasons leading, location, the expected locations where leaks can be prevented. To find the memory leak location and fix them is very useful for application developers to develop high efficiency application.

References

[1] 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.

[2] D. Amal_tano, A. R. Fasolino, P. Tramontana,S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. International Conference on Automated Software Engineering (ASE), pages 258--261, 2012.

[3] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2013.

[4] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In International Conference on MobileSystems, Applications, and Services (MobiSys), pages 239--252, 2011.

[5] A. P. Fuchs, A. Chaudhuri, and J. S. Foster.SCanDroid: Automated security certification of

Android applications. Technical Report CS-TR-4991, University of Maryland, College Park, 2009.

[6]Stack Overflow website: http://stackoverflow.com/questions/tagged/android

[7] 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.

[8] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, “SmartDroid: An automatic system for revealing UI-based trigger conditions in Android applications,” in SPSM, 2012, pp. 93–104.

[9] G. Xu and A. Rountev, “Precise Memory Leak Detection for Java Software Using Container Profiling”, In ICSE, (2008), pp. 151–160.

[10] A. Desnos and G. Gueguen. Android: From reversing to decompilation. In Proc. of Black Hat Abu Dhabi, 2011.