Automated Memory Leakage Detection in Android Based Systems

Jihyun Park and Byoungju Choi[1]

Dept. of Computer Science & Engineering Ewha Womans University, Seoul, Korea
pola0527@ewhain.net, bjchoi@ewha.ac.kr

Abstract

Since open platforms such as Android vary in device manufacturers and application developers, modifications in software happened in multiple layers. Therefore, every layer including OS, library, framework and application may have defects within. Especially, a memory leakage which increases memory usage and diminish overall system performance is the key issue in embedded systems with highly limited resources.

In this paper, we suggest a technique that detects memory leakage by gathering memory execution information in run-time via PCB hooking and apply this technique to actual Android smartphones. The suggested technique does not require a target source code and any hardware changes for memory leakage detection, and it is characterized by maintaining the same target runtime status even in detecting while minimizing performance overhead simultaneously. We implemented an automated tool of this technique for Android Smartphone, and show that it is effective.

Keywords: *Memory Leakage, Android, Software Test*

1. Introduction

Since the embedded system consists of limited hardware resources, the software for the embedded system must support hardware optimization technologies such as lightweight, low-power and efficient resource management. As Android system is embedded in mobile devices such as smartphones, usage of resources (memories and batteries) is limited just like other embedded systems. Therefore, when developing the software, programmer should be able to manage resources efficiently.

Android is a software stack for mobile devices that includes operating system, middleware and applications. Various defects can be found in Android system. Having high frequency of memory leakage like 'Out of Memory' particularly in Android applications, the efficient management of memory usage is needed in Android system. In order to use memory efficiently, memory should be returned to the system after being distributed as required to running programs or applications. However, there is a possibility that programs or applications may not return and keep on occupying unnecessary memory. This program state is called memory leakage.

Memory leakage is a program state that happens either when new memory is being allocated constantly while a program is still running or when a seemingly closed program actually remains in memory. If memory leakage gets worse, it uses up much more memory over time while a program is running. This is a serious problem especially in embedded device where efficient resource usage is important.

In Android system, memory defects occur in different ways between Android application layers operating in virtual machine and processes operating in Linux kernel.

[1] Corresponding author

In Dalvik virtual machine, the heap memory is being allocated for each application and any wasted memory resources are being collected by executing the Garbage Collector periodically.

The Garbage Collector plays a role in removing objects from memory, when the objects generated by an application in execution are no longer in use. However, not knowing exactly when the Garbage Collector is performed, memory shortage may occur if memory is being used unboundedly. In case which a program maintains a reference to an object, even if the object is no longer being used, unnecessary memory waste - memory leakage occurs because the object is not subject to the Garbage Collector[1][2][3].

For processes operating in the Linux kernel, they do not go under any additional memory management. Because of that, there can be various problems such as memory leakage due to not removing memory after its allocation or removing memory after its removal.

In this paper, we suggest an automated memory leakage detection technique considering the characteristics of software in Android platform (hereinafter referred to as "Android") and apply it to actual Android platform for analysis. The suggesting technique does not involve in changing target software code and hardware and is characterized by maintaining the same its runtime status in detecting, and minimizing performance overhead simultaneously.

2. Memory Leakage Detection

There are mainly two causes for memory leakage in Android. The first one is where the object generated in application running on Dalvik virtual machine is not collected by the Garbage Collector, though it is no longer being used and cause memory leakage staying in the memory space of application. The second one is where memory is not returned after being allocated and used in kernel layer. Based on these causes, we suggest an automated memory leakage detection method overcoming restrictions - (1) No modifications to the target software allowed, (2) Maintaining runtime status, (3) Minimizing performance overhead and (4) Targeting all software in the system.

When collecting the data related to memory leakage detection in runtime, optimizing locations where the data are collected is one of the best solutions to minimize system performance degradation. In order to do dynamic embedded software test, we have suggested a test technique that collects test information via hooking process control block (PCB) managing runtime running information in purpose of controlling process by OS [4]. In other words, it is a technique that collects running information of all the processes running in the system via PCB hooking and minimizes corresponding system overhead better than the other test techniques. In this paper, memory leakage detection is automated via PCB hooking.
PCB is a kernel data structure that manages necessary information for controlling specific process. In general, PCB contains information on running processes such as 'process ID, priority, shared library list, specific process-resource list'. Especially, PCB is generated simultaneously as the process is generated, and PCB is always updated with the latest information about the process as long as the process is operated. Since the information gets deleted as the process is terminated, it is the best location for getting the latest information regarding dynamic activities during process running. Our technique has the advantage of minimizing system overload by concentrating detection activity on one location, PCB, rather than spreading over multiple locations.

Hooking is a programming skill that snatches system management and control system in runtime and modifies them into desired direction for specific purpose. In this paper, when a function which becomes the subject to monitoring starts to run, hooking snatches it and run 'Patch function' to collect data regarding memory leakage as in Figure 1. Add to the functions of the preexisting function, the patch function collects data that is for detecting memory leakage.

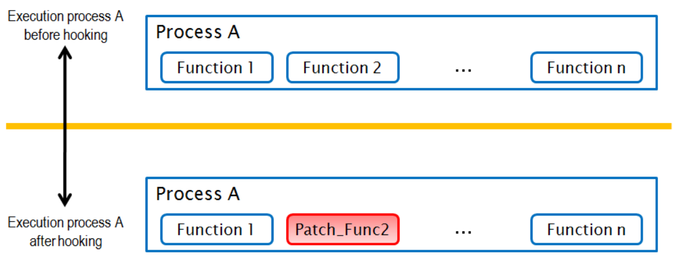


Figure 1. Hooking

Our suggested memory leakage detection technique consists of three steps as follows – identifying PCB, extracting information on memory leakage, and judging memory leakage.

Step 1. Identifying PCB: Identify a kernel memory space where PCB is stored.

Step 2. Extracting information on memory leakage: Extract necessary memory leakage information stored in PCB

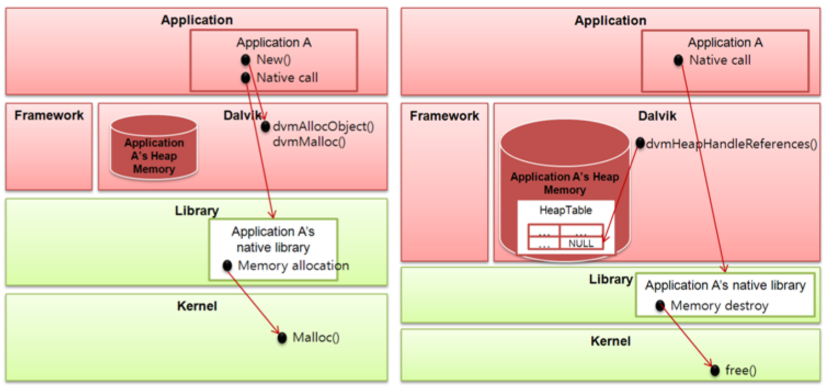
(1) Creating patch function: The patch function that is for extracting memory leakage information is implemented via library being loaded on shared memory space.

(2) Hooking: The location for extracting memory leakage information is identified via PCB and the location gets hooked by the patch function.

(3) Collecting information on memory leakage: When the patch function starts to run, the memory leakage information is collected for defect analysis.

Step 3. Judging memory leakage: Memory leakage is judged by analyzing memory leakage information.

The point is to collect memory leakage related data with patch function from the optimized location for detecting memory leakage from Step 2 in real time.



(a) Memory Allocation (b) Memory Deallocation

Figure 2. Android Memory Management

Unlike general C based programs allocate/remove memory using functions like malloc/free, Android applications is allocated with heap memory space in advance as soon as an object is created as in Figure 2. When an application creates an object, it allocates empty space from allocated heap memory area and increases the reference value of the objects. However, when it no longer uses the object, it decreases the reference value to collect memory later via the Garbage Collector. Therefore, in order to obtain memory leakage information, not only general functions related with memory such as malloc/free, but also the method for allocating the object to heap memory as dvmMalloc/dvmAllocObject is being called when an applicatio n creates an object using 'new', and the dvmHeapHandleReferences method that is being called to check the reference value of the object when the Garbage Collector is performing become the location for detecting memory leakage.

When the hooking-targeted function runs after PCB is being hooked, the patch function starts to perform. For example, as Figure 3 shows, the patch function of dvmMalloc calls the original function of dvmMalloc maintaining the original function of allocating memory for the object, and leaves the necessary data for judging leakage as log form.

```
void* Patch_dvmMalloc(size_t size, int flags) {
   void* return val = dvmMalloc(size, flags);
   writeLog(return_val, size);
   return return_val;
}
```

Figure 3. Patch Function Code of dvmMalloc

In order to judge memory leakage using memory leakage information extracted from Step 2, the address of the allocated object and the address of removed memory. Memory leakage is judged after tracking whether the address allocated to malloc, calloc or realloc is not removed to free and still remains. It is judged as memory leakage as the reference value for the allocated object still remains if the object address allocated to dvmMalloc, dvmAllocObject or dvmHeapHandle References runs.

We implemented it as an automation tool and named it as AMOS for Android (hereinafter referred to as AMOS). AMOS consists of AMOS^{SA} (Scanning Agent) and AMOS^{TM} (Test Manager) as shown in Figure 4. AMOS^{SA} is equipped in the targeted system, and collect information on memory leakage. AMOS^{TM} (Test Manager) analyzes information collected from the binary image and scanning agent of the targeted software in the host system, and derives the test result from the analysis.

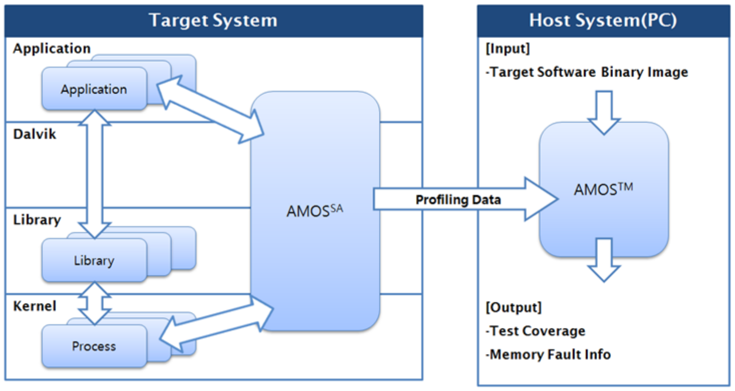


Figure 4. AMOS System Structure

3. Case Study

The smartphones we have used for case study are HTC I/O Device, Linux 2.6.xx and Android 2.2 (froyo). The target applications were 175 applications basically equipped in a Galaxy S smartphone. We organized four test scenarios where memory leakage can occur in applications as follows.

Scenario 1. Creating and terminating an application repeatedly

Scenario 2. Switching between vertical/horizontal views

Scenario 3. UI event occurrence

Scenario 4. Running multiple applications concurrently

We compare our memory leakage detection technique with DDMS and MAT, the memory testing tools for android system, to find out whether our technique can detect memory leakage and analyze the cause of it efficiently[5][6].

When we tested four scenarios to 35 Android applications, memory leakage occurred in total 11 applications as shown in Table 1. For example, in case of the internet web browser, memory leakage occurred, when the test scenarios 1 and 3 are applied. All of the tools such as AMOS, MAT, and DDMS detected leakage.

Table 1. Memory Leakage

Application	Leakage Occurrence	Leakage Detection			Actual Leakage Defects (Reported Leaked Objects)		
	Test Scenario	AMOS	MAT	DDMS	AMOS	MAT	DDMS
V3 Mobile	Scenario 1, 4	Y	Y	Y	2(2)	0(3)	-
Internet Web Browser	Scenario 1, 3	Y	Y	Y	4(4)	1(3)	-
Camera	Scenario 1	Y	Y	Y	3(3)	0(4)	-
Daum Maps	Scenario 1,2,3	Y	Y	Y	11(11)	2(4)	-
E-mail	Scenario 1	Y	Y	Y	6(6)	0(4)	-
Cyworld	Scenario 1	Y	Y	Y	5(5)	0(4)	-
Naver	Scenario 1	Y	Y	Y	7(7)	0(4)	-
Daum TV Pot	Scenario 1	Y	Y	Y	5(5)	0(4)	-
Video Player	Scenario 2	Y	Y	Y	4(4)	2(4)	-
Subway Maps	Scenario 3	Y	N	Y	1(1)	0(0)	-
Naver Maps	Scenario 3	Y	Y	Y	3(3)	2(3)	-
				Total	51(51)	6(37)	-

A leak reporting mechanism is different for each tools and AMOS and MAT shows the object which creates memory leakage. However, not all of the objects shown are the objects that are responsible for the leakage. We will compare the results from the test scenario 1 where it repeats a cycle of creating and terminating the internet web browser 100 times.

We identified detailed information regarding the object by tracking the source code appeared in CallStack information, and the objects responsible for the leakage in the internet web browser are as follows - String object of ‘android.webkit.PluginManager’, char[] object of ‘Android.text.AutoText’, Drawable object of ‘Android.text.AutoText’, and String object of ‘org.apache.http.impl.EnglishReasonPhraseCatalog$DefaultTimeZone’.

These are all actual leakage defects. When the internet web browser is running on the background after being terminated, the objects generated to show the display should be removed and when the application is launched again, new objects should be generated. However, since the internet web browser still refers to the objects which should be removed, the objects do not become the subject of the Garbage Collector and remains in memory, causing memory leakage.

MAT shows the top three classes with the most memory usage as suspicious objects in memory leakage. Regarding this creation-termination repetition test scenario, HashMap object of TrustManagerImpl class, String object of DefaultTimeZones class and Hashtable object of BouncyCastleProvider are appeared as suspicious objects in memory leakage. However, only String object of DefaultTimeZones class is the memory leaking object.

For DDMS, it shows the heap memory usage. When suspecting memory leakage, if the objects are not garbage collected and still remains when a user performs the Garbage Collection using [Cause GC] provided by the tool, and the objects pile up as an application proceeds, it can be judged to be memory leakage. It is possible to know whether memory leakage occurred in an application through this process, however it is impossible to know which objects caused the leakage.

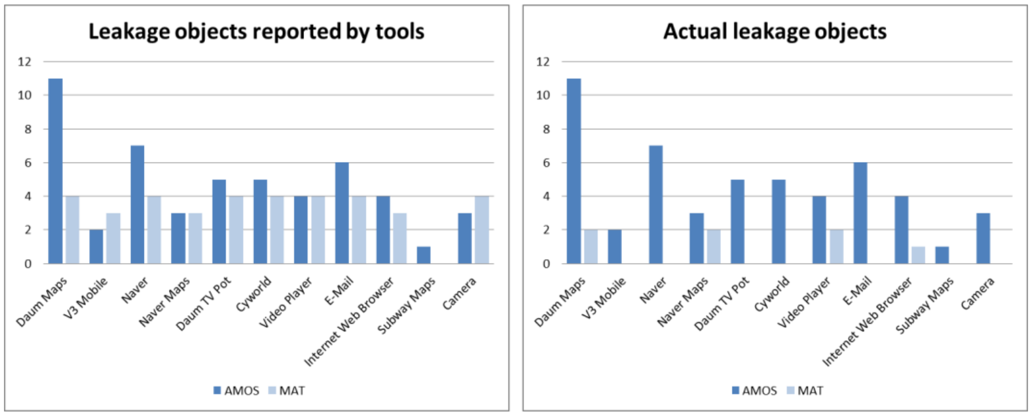


Figure 5. Memory Leakage Test Result Comparison

Figure 5 is the graph of the actual leakage defects vs. objects reported by AMOS and MAT regarding 11 applications where the leakage occurred. According to AMOS, among 51 leakage occurring objects, every objects were the actual defect occurring objects. According to MAT, among 37 reported objects, only six (16%) of them are the actual defect occurring objects.

DDMS and MAT detect leakage by analyzing heap memory of one application. It means that it is possible to detect leakage, when the leakage defect occurred within an application; however, if the leakage is caused by different layers, it is hard to detect the leakage. However, AMOS can test the entire system image to find wherever the leakage occurred. AMOS also can analyze the cause of leakage. AMOS shows CallStack information including names of source code files and functions in order to find the objects responsible for leakage. Analyzing the causes of leakage in 11 applications reveals that the most of the cases are the defects occurred within the applications as shown in Figure 6 and both of the Android Framework layer and kernel had one leakage respectively.

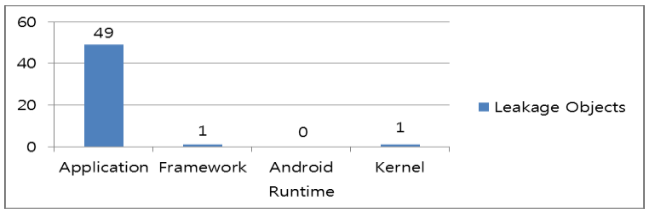


Figure 6. Distribution of Causes of Memory Leakage Defects

The defects occurred in Android applications are mostly the defects where an application maintains the reference to the object due to improper removal of the object, whereas the objects are supposed to be created and removed in process of switching Activity which is the display of an application to foreground and background

In Naver Maps application, memory leakage is caused by LinkedHashMap object generated in 'dalvik.system.PathClassLoader' of Android Framework class. Based on Google Maps, Naver Maps application calls PathClassLoader in order to load Google Maps. We identified that memory leakage occurs as LinkedHashMap generated from PathClassLoader keeps being generated,

The defects occurred in Kernel layer are from Service Administrator Process. As a daemon process which runs as Android boots up, Service Administrator Process registers services into system, and finds proper services when requested by an application. When a new service is generated and the corresponding allocation request enters to Service Administrator Process, the Administrator allocate memory to the request using malloc function, however the defects occur when memory does not get removed, even after the service is terminated.

In AMOS, there is a running agent equipped on a target, and an agent consists of a module, a log collection module, a library defined by hooking functions. Three of these modules take up 64kbytes, which is very small compared to the average size (2.54MB) of applications that come with a Galaxy S. The runtime of an application is also added by 0.14x while the tool is running. However, it barely affects system performance since the overhead of AMOS is smaller than that of DDMS.

Since DDMS and MAT does not have a running module equipped on a target, memory overhead does not occur. For DDMS, the runtime overhead of around 0.174x occurs while tracking heap memory. For MAT, the runtime overhead does not occur because MAT extracts heap log regardless of application execution.

5. Conclusion

Since open platforms such as Android vary in device manufacturers and application developers, modifications in software happened in multiple layers. Therefore, every layer including OS, library, framework and application may have defects within. It is even hard for an application developer to detect defects by checking inside of a system. Even if a developer finds a defect, it is hard for him to debug.

Since memory leakage defects in applications, in particular, increases memory usage and diminish overall system performance, it is the key issue in embedded systems with highly limited resources.

In this paper, we suggested a method that judges memory leakage from a host by tracking memory related functions and real-time logging into memory leakage related data on a target, in order to detect memory leakage defects on Android platform by applying hooking technique. We implemented AMOS and applied to 35 applications on the Android platform and analyzed the result. As a result, 51 memory leakage objects from 11 applications were found. Defects found by AMOS can track the location by log analysis. It was very helpful for debugging on android system and applications.

Applying the same principle to DDMS and MAT, we compared the results with that of AMOS. DDMS only shows heap memory usage and leave a judgment on the memory leakage to user's discretion. On the other hand, MAT and AMOS judges the memory leakage from a tool. MAT analyzes heap memory situation when extracting log, and shows information which turns into a cause of memory leakage defect. AMOS shows the result of memory leakage judgment by analyzing logs collected from the agent. Though, both of the tools inform the causes of memory leakage, the accuracy of MAT is 18% which is very low compared to 100% accuracy of AMOS.

Based on the findings of this study, we are now expanding our work to detect the defects occurring from interactions between Android applications.

Acknowledgements

This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency(NIPA-2012-(H0301-12-3004))).

References

[1] M. Jump, K. S. McKinley, “Cork: Dynamic Memory Leak Detection for Garbage-Collected Languages”, SIGPLAN Not., vol. 42, **(2007)**, pp. 31-38.

[2] Y. Tang, Q. Gao and F. Qin, “LeakSurvivor: Towards Safely Tolerating Memory Leaks for Garbage-Collected Language”, In USENIX Annual Technical Conference, **(2008)**, pp. 307–320.

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

[4] J. Seo, B. Choi and S. Yang, "A Profiling Method by PCB Hooking and Its Application for Memory Fault Detection in Embedded System Operational Test", Journal of Information and Software Technology, vol. 53, no. 1, **(2011)**, pp. 106-117.

[5] DDMS, http://developer.android.com/guide/developing/debugging/ddms.html.

[6] MAT, http://psychcorp.pearsonassessments.com/haiweb/Cultures/en-US/site/Community/PostSecondary/Products/MAT/mathome.htm.

Authors

Jihyun Park

Jihyun Park is a Ph.D. student in the Department of Computer Science and Engineering at Ewha Womans University in Korea. Park holds B.S. and M.S. degrees in Computer Scicence and Engineering from Ewha Womans University.

Byoungju Choi

Byoungju Choi is a full professor in the Department of Computer Science and Engineering at Ewha Womans University in Korea. Choi holds a B.S. degree in mathematics from Ewha Womans University, M.S. and Ph.D. degrees in computer sciences from Perdue University, USA. Choi’s research interests include software engineering with particular emphasis on software testing, risk based testing, embedded software testing, and software process improvement.