Framework of Analysis Technique for Abnormal Behavior in Mobile Application (FATABMA)

NAQLIYAH BT ZAINUDDIN*, MOHD.FAIZAL BIN ABDOLLAH,ROBIAH BT YUSOF,SHAHRIN BIN SAHIB
Faculty of Information and Communication Technology
University Technical Malaysia Malacca
Karung Berkunci No. 1752 Pejabat Pos Durian Tunggal, 76109 Melaka
MALAYSIA
*M031110026@student.utem.edu.my
[faizalabdollah,robiah,shahrin]@utem.edu.my
http://www.utem.edu.my

Abstract: - Abnormal application behavior in mobile can produce a number of undesirable effects. An incorrect or insufficient implementation of application lifecycle, memory related issues and malicious application might cause an unexpected behavior of the application such as bad usability, not responding, crashed and even data loss. Current analysis and detection of abnormal applications behavior are still not comprehensive enough where behavior under user visible failure category such as crash, “stopped unexpectedly” and “not responding” received less attention by researchers. Furthermore, framework of analysis technique has not been developed by researcher to investigate the abnormal behavior in mobile application. Thus, this research will introduce the framework of analysis technique for abnormal behavior in mobile application. In this paper, both static and dynamic analysis techniques are described and applied to Android applications to identify causes of abnormal behavior. These allow the identification of android abnormal application behavior into “behavior groups”, which consists certain behavior that tends to have similar generalized activity profiles.

Key-Words: Abnormal behavior, application, Android, analysis techniques

1 Introduction

In today’s world, mobile applications are becoming increasingly important in all aspects of our lives. No longer are phones reserved just for making calls, they now do more than the PC’s of a few years ago. The open source Android operating system is a great example of the future of mobile applications. The rapid growth of smartphones has lead to a renaissance in mobile application services. Android and iOS, currently the most popular smartphone platforms, each offer their own public marketplace.

Detection of malwares, resources issues and others factors which causing unexpected or abnormal behavior in mobile application has been the main focus by researchers in mobile security. In the area of mobile applications, an application life cycle plays an important role. According to Lbishop (2012), Android applications must conform to the Android OS’s concepts of application lifecycle. Franke *et. al.* (2012) also highlighted that, an incorrect or insufficient implementation of the life cycle might cause unexpected behavior of the application, leading to bad usability and even data loss. When an application crashes, it may disrupts the user experience, cause data loss, and worst of all, might even cause users to uninstall the application altogether.

Malicious software also will result in unexpected behavior by attempting to leak personal information, getting root privilege and abuse functions of the mobile [3]. Luo *et. al.* (2012) has stressed that even if applications have acquired explicit user consents, users may be unaware that the applications may execute malicious behaviors. Besides, Delac *et. al.*

(2011) also highlighted other standard malicious attacks for PCs, like worms and Trojans is also becoming applicable to the mobile platforms. Malicious software such as *Geimini* and *DroidDream* will result in unexpected behavior by attempting to leak personal information, getting root privilege and abuse functions of the smartphone as reported by Isohara *et. al.* (2011a). Pocatilu (2011) also has reported that the behavior of malicious applications could vary from annoying messages to very unrecoverable damages. Definitely, a compromised smartphone can inflict severe damages and caused unexpected behavior in Android application. Moreover, memory leaks also are highlighted by Joshi (2012) as one of the major issues seen on the performance side of the mobile application which causing a sluggish behavior. Peng *et. al.* (2008) and Park *et. al.* (2012) also emphasized that the memory leak phenomenon will affect the memory usage, affects the application to switch efficiency and cause the increase of memory usage and diminish overall system performance. Despite the capability of Android to handle memory allocation using garbage collection, Shahriar *et. al.*(2014) in his research identified that many applications currently suffer from memory leak vulnerabilities and causing applications to crash due to out of memory error while running.

Above researchers highlighted the possible reasons on unexpected or abnormal behavior in an android application. Despite the outbreak of research activity in this area, Wei (2013) has highlighted that there is no framework yet that focuses on analysis and profiling the behavior of an Android application. Definitely, abnormal behavior in mobile application can produce a number of undesirable effects which might cause an unexpected behavior such as bad usability, not responding, crashed and even data loss. Majority of works done are focusing on detecting of malicious behavior due to malicious software whereas less work done so far in identifying abnormal application behavior which causing application to crash, “stopped unexpectedly” and “not responding”.

2 Related Works

“CrowDroid” is a framework introduced by Burguera *et. al.* (2011b). The framework is using dynamic analysis on system call (*Strace*) which enable the distinguishing between applications that having the same name and version but behave differently. The focus of the framework is to detect anomalously application in form of Trojan horses. CrowDroid used *Strace* to output the behavior patterns such as system calls of installed applications on users' devices. This information is sent to a remote server where the system calls are clustered using a K-means algorithm into benign and malicious categories. CrowDroid concluded that *open()*, *read()*, *access()*, *chmod()* and *chown()* are the most used system calls by malware. Moreover, Shabtai *et. al.* (2011) introduced “Andromaly” another behavioral malware detection framework for android devices. Andromaly is a lightweight malware detection system using Machine Learning classification techniques to classify collected observations (system performance, user activity, memory, CPU consumption, battery exhaustion etc) as either normal or abnormal.

Another work is by Bl *et. al.* (2010) proposed “AASandbox” (Android Application Sandbox). AASandbox is using static and dynamic approach to automatically detect suspicious application. For static approach, AASandbox scans the software for malicious patterns without installing it. While for dynamic approach, the analysis on the application is conducted in fully isolated environment which intervenes and logs low-level interactions. Karami *et. al.*(2013) had introduced a comprehensive software inspection framework. The framework allows identification of software reliability flaws and to trigger malware without require source-code. The framework is using dynamic approach by collecting run-time behavior analysis and also the *I/O* system calls generated by the applications.

Kim (2011) had introduced “ModelZ” for monitoring, detection, and analysis of energy-greedy anomalies in mobile handsets. Using light weight approach, ModelZ will monitor, detect and analyze new or unknown threats and energy-greedy anomalies on small mobile devices, with high accuracy and efficiency. Alazab *et. al.*(2012) introduced "DroidBox" a dynamic analysis tool to classify Android applications by monitoring API calls of interest invoked by an application. The analysis includes generating two graphs (behavior graphs and treemap graphs) for sample in order to provide the basis in identifying benign or malicious categories.

Thing *et. al.*(2011) also had used system call, logs and timestamp information in his research to detect the "misbehaving" applications, alert the users, and log the evidence of malicious activities with. From the discussion on analysis technique in detecting malicious application, *Strace* is identified as a common tool in Android research and it has been used in works on malware detection by most of the researchers. *Strace* used the view of Linux-kernel such as network traffic, system calls, and file system logs to detect anomalies in the Android system. Furthermore, Burguera *et. al.* (2011a) also emphasized that monitoring systems calls (*Strace*) is one of the most accurate techniques to determine the behavior of an Android application since they provide detailed low level information. In the next section, we will discuss on other analysis technique used by researchers in analyzing other type of abnormal behavior due to resources leaks and application life-cycle.

The detection of resources problems in mobile application has been studied by Guo *et. al.* (2013), Yan (2013) and Park *et. al.* (2012). Guo *et. al.* (2013) introduced an approach using static analysis tools called *Relda*, which can automatically analyze an application's resource operations and locate resource leaks. The method is based on a modified Function Call Graph, which handles the features of event-driven mobile programming by analyzing the callbacks defined in Android framework.

Yan (2013) proposed a novel and comprehensive approach for systematic testing for resource leaks in Android application. The approach is based on a GUI model, but is focused specifically on coverage criteria aimed at resource leak defects. These criteria are based on neutral cycles: sequences of GUI events that should have a "neutral" effect and should not lead to increases in resource usage.

The work on memory leakage detection is by Park *et. al.* (2012b) using a PCB hooking technique. The technique is using dynamic analysis by gathering memory execution information (i.e.; process ID, priority, shared library list, specific process-resource list) in run-time to detect memory leakage. In the experiment, Memory Analysis Tool (MAT) was used as a comparison with their invented tool.

The only work on monitoring software crashes is by Kim *et. al.* (2010) who presented a framework which monitors and reproduces software crashes. This approach involves learning patterns from features of methods that previously crashed to classify new methods as crash-prone or crash-resistant. Investigations had shown that 30% of crashed methods in ECLIPSE and 44% from ASPECTJ threw exceptions. The remaining 70% of crashed methods are not throwable and it is less common to see developers throw runtime exceptions in their programs.

Futhermore, Franke *et. al.* (2012) presented a tool called 'AndroLIFT' which helps the developer to monitor the life cycle, assists in implementing it and testing life cycle-related properties. AndroLIFT is written as an extension to the ADT, the common way of developing Android applications with the Eclipse IDE. The life cycle view of this tool allows the developer to observe and analyze the life cycle of the Android application. Besides, it allow developer to easily learn about the behavior of the application life cycle to certain triggers, like an incoming call, and with which callback methods one can react appropriately. The summary of analysis techniques used in the detecting malicious and abnormality in mobile application is depicted in Table 2.1.

Table 2.1: Analysis Techniques in Detecting Abnormal Behavior in Mobile Application

Works Related	Category	Criteria of Detecting Abnormal Behavior
ModelZ Kim (2011)	Energy-greedy anomalies	Monitor and record usage of software and hardware resouces
CrowDroid Burguera *et.*	Malicious software	Using *Strace* to output the behavior patterns such as

Works Related	Category	Criteria of Detecting Abnormal Behavior
al. (2011)		system calls
Andromaly Shabtai *et. al* (2011)	Malicious software	Using Machine Learning Classification to classify collected observation information
AASandbox Bl *et. al.* (2010)	Malicious software	Intervenes and logs low-level interaction of an apps
Karami *et. al.* (2013)	Malicious software	Collecting run-time behavior analysis and also I/O system calls generated by an apps
Isohara *et. al.* (2011b)	Malicious software	Using log collector to record activity on kernel layer
Guo *et. al.* (2013)	Resource leaks	Using Function Call Graph
Yan (2013)	Resource leaks	Using GUI model to detect resource leaks defect
Park *et. al.* (2012)	Memory leakage	Using PCB hooking technique to gather memory execution information
Kim *et. al.* (2010)	Crash method	Learning patterns from features of the method that previously crashed
AndroLIFT Franke *et. al.* (2012)	Monitor apps life-cycle	Using an extension to ADT
DroidBox Alazab *et. al.* (2012)	Malicious software	Monitoring API calls of interest invoked by an apps
Thing *et. al.* (2011)	Malicious software	Using *strace* to log the system call, logs and timestamps information invoked by an apps
Wei (2013)	Profiling of Android application	Measure and profile the apps at four layers

All in all, the aforementioned frameworks and systems as stated in Table 2.1 proved valuable in protecting mobile devices in general. Most of the works are focusing on malware detection in mobile application using both dynamic and static analysis techniques. Detection technique on malicious software received a lot of attention by researchers. However, there is a gap in identifying the abnormal behavior which may lead to behavior of crash, “stopped unexpectedly” and “not responding”. Therefore this research will develop a framework of analysis technique for abnormal behavior for user visible failure category which includes crash, “stopped unexpectedly” and “not responding”.

From the discussion on analysis technique in detecting malicious application, *Strace* is identified as a common tool in Android research and it has been used in works on malware detection by most of the researchers. *Strace* used the view of Linux-kernel such as network traffic, system calls, and file system logs to detect anomalies in the Android system. Furthermore, Burguera *et. al.* (2011a) also emphasized that monitoring systems calls (*Strace*) is one of the most accurate techniques to determine the behavior of an Android application since they provide detailed low level information.

Moreover, Franke *et. al.* (2012) has highlighted that logcat is identified as the main logging mechanism in mobile application. Logcat allows us to capture the system debug output and log messages from the application. Wei (2013) used a combination of the *logcat* and *getevent* tools of ADB to gather the data of the user layer for multi-layer profiling of Android application.

A specific tool for memory analysis is Memory Analyzer Tool (MAT). The *MAT* tooling is a set of plug-ins which visualizes the references to objects based on Java *heap dumps* and provides tools to identify potential memory leaks in Android applications. *The MAT* detects leakage by analyzing heap memory of one application. MAT analyzes heap memory situation when extracting log, and shows information which turns into a cause of memory leakage defect (Park *et. al.*, 2012).

Therefore, the proposed framework of analysis technique here is adapting the techniques used by existing researchers in mobile application research. As shown in Table 2.2, this research utilized a combination of Linux trace (*Strace*) and Android debug facilities techniques (*logcat* and MAT) to profile the abnormal behavior in mobile application for user visible failure category which are crash, “stopped unexpectedly” and “not responding”.

Table 2.2: Analysis Techniques for Abnormal Behavior

3 Datasets and Environment

Applications involved in these experiments are from various categories such as games, security software, tools and some others that were randomly installed from Google Play and other sites such as from http://www.appsapk.com/ and http://www.androidcentral.com/apps. Using APK Downloader which runs under Google Chrome, an .apk files was downloaded into laptop prior to configuration under emulator.

For emulator, the logs were captured using logcat, a logging application in Android platform tools. The analysis also includes logs gathered from other researcher who also conducted some experiment on Android platform. Total logs size are more than 3GB. However, crashed logs are found less than 10%. The focus of this analysis is on abnormal application behavior and these findings were later analyzed to construct a framework for abnormal application behavior. The configurations of the devices is shown in Table 3.1.

Dataset	Devices	Configurations	Total Apps.
1	Galaxy Tablet Samsung P3100 4.0.4	Total Space 11GB	120
2	Android Emulator 2.3.4 (API Level 10)	RAM=343, VM Heap = 32, Internal Storage=200 MiB	20
	Android Emulator 2.3.3 (API Level 10)	RAM=343, VM Heap = 32, Internal Storage=200 MiB	40
3	Android Emulator 2.2 (API Level 8)	RAM=512, VM Heap = 16, Internal Storage=200 MiB	50

Table 3.1: Devices Cofigurations

For simulation on the emulator, we organized three test scenarios where possible abnormal behavior such as; memory leakage can occur in applications. The scenarios are as follows.

- Scenario 1- Running multiple applications simultaneously

No	Techniques	Tools/Interface	Objectives
1	Logcat Analysis	Manual analysis on extracted stack traces	To understand the application level activity sequence for abnormal activity
2	Heap dump Analysis	Eclipse MAT (Memory Analyzer Tool)	To identify the objects and classes consuming memory in the java *heap*
3	STRACE Analysis	adb shell strace -p <PID_number>	To identify system calls or signals made to the OS

- Scenario 2-Switching between vertical/horizontal views
- Scenario 3- Repeatedly creating and terminating an application

These similar test scenarios were also conducted by Park *et. al.* (2012b) in Android memory related experiments. On real devices, around 120 applications were installed and monitored for abnormal application behavior. *The logs were col*lected using an application named *LogCollector.apk* and also using automated crashed detector application named *Crash Log.apk* in the event of application is crashing, not responding or unexpectedly stopped.

The framework for analysis technique as shown in Figure 3.1 is proposed as this framework is needed for identifying abnormal behavior in mobile application.

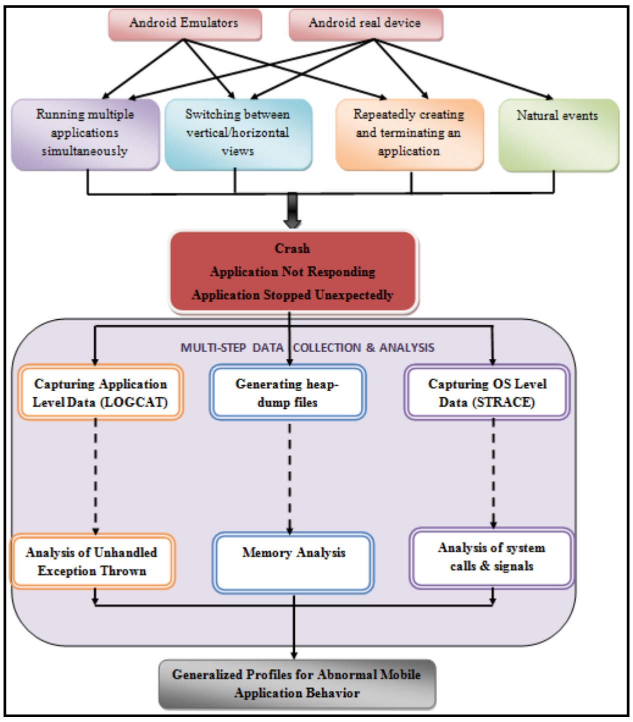

Android Emulators
Android real device
Running multiple applications simultaneously
Switching between vertical/horizontal views
Repeatedly creating and terminating an application
Natural events
Crash
Application Not Responding
Application Stopped Unexpectedly
MULTI-STEP DATA COLLECTION & ANALYSIS
Capturing Application Level Data (LOGCAT)
Generating heap-dump files
Capturing OS Level Data (STRACE)
Analysis of Unhandled Exception Thrown
Memory Analysis
Analysis of system calls & signals
Generalized Profiles for Abnormal Mobile Application Behavior

Figure 3.1: Framework for Analysis Technique of Abnormal Behavior in Mobile Application (FATABMA)

4 Analysis and Results

Based on framework proposed above, 3 multistep data collection and analysis are conducted as following:

4.1 Logcat Analysis

For logcat analysis, we have collected around 50 stack traces for crashes and ANR applications. We analyzed the abnormal behavior manually based on keywords identified. Details analysis of the attributes gathered from the logcat are as following:

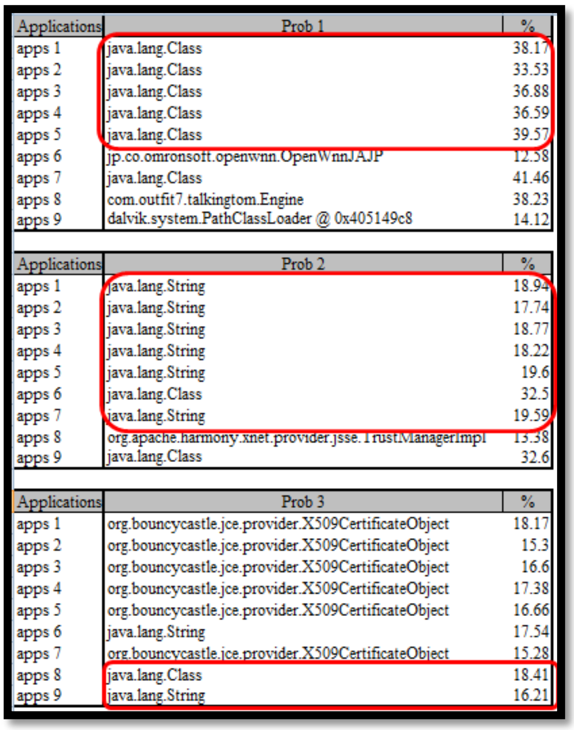
a) Identification of warnings/errors thrown
The result had shown that 80% of the crashes and ANR thrown an unhandled exception due to *java.lang*. The unhandled exceptions thrown include *illegalArgumentException*, *illegalStateException*, n*ullPointerException*, *outOfMemory* and other exceptions. Memory related problems can be identified from the extracted stack traces such as *java.lang.OutOfMemoryError*, *buffer overflow* and *Out of memory*.

b) Identification of GC activity patterns
There are 5 types of GC can be found in *logcat*. GC_CONCURRENT is triggered when a heap is growing by reclaiming memory in time so that the heap does not need to be enlarged. GC_EXPLICIT is triggered when an application issuing *System.gc()* method. Since the virtual machine is quite capable of handling GC, this type of GC should actually never be called. GC_EXTERNAL_MALLOC is used for externally allocated memory like bitmaps and NIO direct byte buffers. This is only on pre-Honeycomb devices because from Honeycomb the external memory is allocated inside dalvik heap. GC_FOR_MALLOC is triggered when the heap is full and the application needs more memory. This will stop application to perform a GC. GC_HPROF_DUMP_HEAP is triggered when an HPROF file is created for memory analysis. GC activities in this context are an automatic and continuous process until the application abnormally terminated. GC_Concurrent and GC_For_Malloc are occurred automatically and repeatedly. GC_For_Malloc was triggered when there was not enough memory left on the heap to perform an allocation. While, GC_Concurrent was triggered when the heap has reached a certain amount of objects to collect and appeared to free <1K left memory. Therefore, this research concluded that GCForAlloc is the most frequent GC appeared in this study which is 68%.

c) Identification of CPU Usage patterns
We found very little logs thrown the CPU usage indicators for their applications. However, the result shows an increase in CPU usage more than 40% for ANR and crashed applications. An application is totally crashed or not responding when the log throws the messages such as “service crashed”, “*WINDOW died*” and “*WIN death*”. Initial analysis using *logcat* allows us to investigate on possibility of memory issues on application.

4.2 Heap-dump Analysis

Heap dump files were later generated from crashed applications to perform the memory analysis. In this phase, we managed to capture 40 crashed applications with the heap size around 1.4MB to 4.8MB. The results show that all suspected leaks are on the same classes/objects which are; *java.lang.Class* (leaks suspects range around 33% - 39%), *java.lang.String* (leaks suspects range around 17% - 19%) and *org.bouncycastle.jce.provider.X509Certific ateObject* (leaks suspects range around 15% - 17%). Highest memory consumptions by objects/classes are found on *org.apache.harmony.xnet.provider* (around 186KB), class *android.text* (around 123KB), and *org.bouncycastle.jce* (around 61KB).



Applications	Prob 1	%
apps 1	java.lang.Class	38.17
apps 2	java.lang.Class	33.53
apps 3	java.lang.Class	36.88
apps 4	java.lang.Class	36.59
apps 5	java.lang.Class	39.57
apps 6	jp.co.omronsoft.openwnn.OpenWnnJAJP	12.58
apps 7	java.lang.Class	41.46
apps 8	com.outfit7.talkingtom.Engine	38.23
apps 9	dalvik.system.PathClassLoader @ 0x405149c8	14.12

Applications	Prob 2	%
apps 1	java.lang.String	18.94
apps 2	java.lang.String	17.74
apps 3	java.lang.String	18.77
apps 4	java.lang.String	18.22
apps 5	java.lang.String	19.6
apps 6	java.lang.Class	32.5
apps 7	java.lang.String	19.59
apps 8	org.apache.harmony.xnet.provider.jsse.TrustManagerImpl	13.38
apps 9	java.lang.Class	32.6

Applications	Prob 3	%
apps 1	org.bouncycastle.jce.provider.X509CertificateObject	18.17
apps 2	org.bouncycastle.jce.provider.X509CertificateObject	15.3
apps 3	org.bouncycastle.jce.provider.X509CertificateObject	16.6
apps 4	org.bouncycastle.jce.provider.X509CertificateObject	17.38
apps 5	org.bouncycastle.jce.provider.X509CertificateObject	16.66
apps 6	java.lang.String	17.54
apps 7	org.bouncycastle.jce.provider.X509CertificateObject	15.28
apps 8	java.lang.Class	18.41
apps 9	java.lang.String	16.21

Figure 4.1: Leaks Suspected on Crashed/ANR Applications

Figure 4.1 shows only some portion of applications with leaks suspected objects and classes. So far, leaks suspected and high memory consumptions are found on the same classes/object.

4.3 STRACE

The result shows that the only system call made during ANR and crashed application is "SIGKILL". The *strace* will return a message "*kill(<pid>, SIGKILL <unfinished ...>*" to terminate the process immediately. In Linux, the *kill* command is used to terminate processes without having to log out or *reboot* (i.e., restart) the computer or devices. A *pid* is a unique process identification number belonging to each process and it is created to be used by the system for referencing to the process. This is to ensure the stability of such systems, in other words to ensure that an application is completely terminated. No other system call was found for this type of behavior.

5 Conclusions

The framework of analysis techniques for abnormal application behavior is presented here as a way to identify the reasons of abnormal activity in mobile application This work proposes a framework of analysis technique on identifying abnormal behavior patterns: (i) To understand the application level activity sequences for abnormal activity via *logcat* (ii) To identify the objects and classes consuming memory in the java *heap* (iii) To identify system calls or signals made to the OS using *Strace*. This research discovered common patterns by applications in category user visible failure which are; crash, "stopped unexpectedly" and "not responding" where stack trace analysis provide initial information for further investigation on possibility of memory related issues in an application.

REFERENCES

[1] Lbishop, "ANDROID LIFECYCLE FOR APPLICATION DEVELOPERS□:," no. May, 2012.

[2] D. Franke and T. Roy, "AndroLIFT□: A Tool for Android Application Life Cycles," no. c, pp. 28–33, 2012.

[3] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based Behavior Analysis for Android Malware Detection," pp. 1011–1015, 2011.

[4] H. Luo, G. He, X. Lin, and X. (Sherman) Shen, "Towards hierarchical security framework for smartphones," *2012 1st IEEE Int. Conf. Commun. China*, pp. 214–219, Aug. 2012.

[5] G. Delac, M. Silic, and J. Krolo, "Emerging Security Threats for Mobile Platforms," *Electr. Eng.*, pp. 1468–1473, 2011.

[6] P. Pocatilu, "Android Applications Security," *Management*, vol. 15, no. 3, pp. 163–172, 2011.

[7] M. Joshi, “Analysis and Debugging of OEM’s,” 2012.

[8] L. Peng, J.-K. Peir, T. K. Prakash, C. Staelin, Y.-K. Chen, and D. Koppelman, “Memory hierarchy performance measurement of commercial dual-core desktop processors,” *J. Syst. Archit.*, vol. 54, no. 8, pp. 816–828, Aug. 2008.

[9] J. Park and B. Choi, “Automated Memory Leakage Detection in Android Based Systems,” vol. 5, no. 2, pp. 35–42, 2012.

[10] H. Shahriar, S. North, and E. Mawangi, “Testing of Memory Leak in Android Applications,” *2014 IEEE 15th Int. Symp. High-Assurance Syst. Eng.*, pp. 176–183, Jan. 2014.

[11] X. Wei, “ProfileDroid□: Multi-layer Profiling of Android Applications Categories and Subject Descriptors,” 2013.

[12] I. Burguera and U. Zurutuza, “Crowdroid□: Behavior-Based Malware Detection System for Android,” *System*, pp. 15–25, 2011.

[13] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, “‘Andromaly’: a behavioral malware detection framework for android devices,” *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, Jan. 2011.

[14] T. Bl, L. Batyuk, A. Schmidt, S. A. Camtepe, S. Albayrak, and T. Universit, “An Android Application Sandbox System for Suspicious Software Detection,” *Techniques*, pp. 55–62, 2010.

[15] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou, “Behavioral Analysis of Android Applications Using Automated Instrumentation,” *2013 IEEE Seventh Int. Conf. Softw. Secur. Reliab. Companion*, pp. 182–187, Jun. 2013.

[16] H. Kim, “MODELZ: Monitoring, Detection, and Analysis of Energy-Greedy Anomalies in Mobile Handsets,” *IEEE Trans. Mob. Comput.*, vol. 10, no. 7, pp. 968–981, Jul. 2011.

[17] M. Alazab, V. Monsamy, L. Batten, P. Lantz, and R. Tian, “Analysis of Malicious and Benign Android Applications,” *2012 32nd Int. Conf. Distrib. Comput. Syst. Work.*, pp. 608–616, Jun. 2012.

[18] V. L. L. Thing, P. P. Subramaniam, F. S. Tsai, and T. Chua, “Mobile Phone Anomalous Behaviour Detection for Real-time Information Theft Tracking,” no. c, pp. 7–11, 2011.

[19] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, “Characterizing and detecting resource leaks in Android applications,” *2013 28th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pp. 389–398, Nov. 2013.

[20] D. Yan, “Systematic Testing for Resource Leaks in Android Applications,” no. Vm, pp. 411–420, 2013.

[21] S. Kim, N. Bettenburg, and T. Zimmermann, “Predicting Method Crashes,” 2010.

[22] T. Isohara, K. Takemori, and A. Kubota, “Kernel-based Behavior Analysis for Android Malware Detection,” *2011 Seventh Int. Conf. Comput. Intell. Secur.*, pp. 1011–1015, Dec. 2011.