An empirical study of software aging manifestations in Android

Yu Qiao
Department of Automatic Control,
Beihang University, Beijing, China
Email: qiaoyu@buaa.edu.cn

Zheng Zheng*
Department of Automatic Control,
Beihang University, Beijing, China
Email: zhengz@buaa.edu.cn

FangYun Qin
Department of Automatic Control,
Beihang University, Beijing, China
Email: fangyunqin@buaa.edu.cn

***Abstract*—Software aging phenomenon has been widely observed in Android mobile operating system, which will make a great influence on people's life. In this paper we present an empirical study on the manifestations of aging-related bugs in Android operating system. By conducting a set of experiments through building various aging conditions in Android, we clarify the fact that the aging-related bugs injected in Android are with several manifestation patterns from two angles of view (Users' and System's viewpoints), depending on their location (Dalvik Heap or Native Heap) , process priorities (cached, persistent, background, foreground and etc.). To show the new features of software aging in Android, the differences of software aging manifestations between Android and Linux are further explored in this work.**

***Index Terms*—software aging; Android; low memory killer; response time**

I. Introduction

Android is an open-source operating system that has a growing influence on human being's lives due to its close relationship with smartphones in current days. It is developed based on Linux and mainly designed for touchscreen mobile devices such as smartphones, tablets and etc. Various software aging phenomena have been observed in the bug reports of Android operating system, in which unsmooth, unexpected block, and suddenly crash of an application after long-time running due to aging-related bugs are typical examples [1]. Therefore, it is significant to study software aging manifestations in Android to help developers keep effective control of their applications, prevent or eliminate performance degradation, enable the immediate response to solve the issues related to memory consumption and avoid unexpected failures on the system[2], [3], [4], [5].

Many studies have been taken on software aging in long-time running operating systems. Most researches focus on Unix and Linux systems [6], [7], while a nonnegligible part on other OSs include Solaris [8], Windows NT [9] and Space systems [10], [11] were also studied. Although Android is developed based on Linux, there are many differences between them. Besides the components like ashmen, wakelocks, LMK (low memory killer), logger on kernel layer, Android uses DVM (Dalvik virtual machine) as its runtime environment in most of their versions(before 5.0), and with JIT (just-in-time) compilation technique inside. Besides, Android has its own specific standard C library-Bionic, above which it builds Android application framework and Android applications, mainly written in Java. In Android, the connection between Java and C/C++ is through the usage of JNI (Java Native Interface).

The differences between Android and Linux lead to the following challenges for the study of software aging in Android:

(1) New software aging manifestation caused by the utilization of DVM and JNI. The usage of DVM and JNI makes two heap areas (Dalvik Heap and Native Heap) exist in Android. Davlik Heap is limited by its own maximum size, memory leak in which will trigger GC (Garbage Collection) frequently and lead to suspend of other threads, while the size of Native Heap is limited by the whole system memory and GC will not take charge of it. This will lead to different manifestations for memory leak in different heap areas both from users' and system's viewpoints.

(2) New software aging manifestation caused by the utilization of LMK. Based on the working mechanism of LMK, software aging phenomenon due to the memory exhaustion caused by processes owing different priorities will have different manifestations from both users' and system's viewpoints, which leads to the complexity of analyzing software aging in Android.

To resolve above issues, we conducted a set of experiments to verify aging manifestations under various aging conditions from users' and system's viewpoints, and discussed their differences with the aging indicators in Linux. The main findings in our empirical study are briefly shown in Table I.

The rest of this paper is organized as follows. Section II presents Research questions and the experimental setups and plans, followed by its results in section III. Conclusions and future work are discussed in Section IV.

II. Empirical study

A. Research questions

In this work we focus on the following two research questions according to the challenges for the study of software aging in Android proposed in Section 1.

RQ1: How will software aging in Android manifest from the viewpoint of users under various aging conditions?

RQ2: How will software aging manifest from system's viewpoint under various aging conditions and what are the differences with Linux?

978-1-5090-3601-1/16 $31.00 © 2016 IEEE
DOI 10.1109/ISSREW.2016.19

TABLE I: Research results

	Dalvik Heap leak	Native Heap leak in cached process	Native Heap leak in persistent process
Aging manifestation to users	An increasing trend in general and frequently fluctuations in local area of the response time as time goes by; A crash of an application after long-time running	No significant influence	An increasing of response time for users' actions after long-time running, which will last for a long time
Aging manifestation to system	Decreasing trend of the indicator RealFree; Increasing trend of Dalvik Heap Size and Heap Utilization	Decreasing trend of RealFree and makes little available memory left; It can be self-recover	Decreasing trend of RealFree and makes little available memory left; It can be self-recover

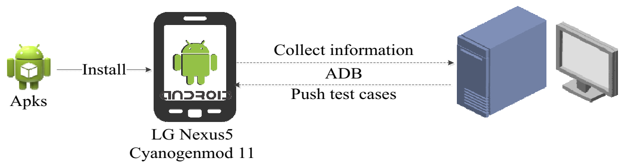


Fig. 1: Experiment environment

To answer the questions, we take an empirical study to observe the manifestations of software aging from the two viewpoints by collecting the memory usage information and the launching time of tested applications. Through the analysis of these data, the differences with Linux are also discussed.

B. Experiment setup

This section presents the experiment setup for the empirical study. Fig.1 presents a brief view of the experiment environment and we will elaborate on the key parts in the following.

1) Hardware and Software: All of the experiments are taken on LG Nexus5 mobile phone equipped with 2GB of RAM, and 16GB internal storage. The mobile phone is installed with an Android 4.4 operating system. The specific version of the Android system is Cyanogenmod 11.1, which is developed based on Android 4.4 KitKat. We use Android 4.4 as the main objective because it is the most popular version of Android[12] and the memory management scheme in Android does not vary much in different versions.

A computer is used to feed test cases into the mobile phone and collect the required information. It contains 2GB RAM, and 500GB hard disk. The operating system installed on the computer is 64-bit Ubuntu 14.04.

2) User task simulator: We use the API package provided by *UI Automator* [13] and develop a program to simulate user tasks in our experiments, such as taking camera, sending email, calling phones and etc.

3) Data collection and ADB: The collected data consists of two types: one is the system state information such as memory, disk, current and etc., which are mainly caught from virtual file system /proc (Type I in short). The information inherits from Linux. The other is Android specific ones, such as Apks memory information, process information and etc. (Type II in short). In our work, we use the tools like *procrank*, *dumpsys*, and *logcat* [14] provided by Android to collect the second type of information.

All of above information are gathered through *ADB* (Android Debug Bridge), a command line tool to handle the communication between computer and smartphone.

In our experiment, the information to measure the response time is gathered from the log of Android. Through the collection of log messages, we extract an application's launching time to represent its response time to users' requirements of opening an application.

4) Apks: Two applications are developed by us to initialize the operating system and help to build the complex aging conditions. The first one is *Pre-condition.apk*. It initializes the phone state with predefined numbers of messages, contacts, and called phones.The second one is *Insert-native-memory.apk*. It helps inject memory leak in Native Heap.

C. Experimental plan

To ensure the experiments run on the same initial condition, all the experiments are taken after backup and reset the phone. In this work, the camera application and the calculator application are selected as the main objectives. From Android OS's viewpoint, each app is transformed to one or several processes in the OS. Thus our experimental results are not very sensitive to the selection rule of specific apps.

In this work, we define normal conditions as the original state of Android applications with no memory leak injected. User perception is mainly measured by the response time when users open an application or undertake other operations, and the influence of software aging on the system are mainly measured by the trend of memory related information.

We conducted five experiments, and to clearly distinguish the experiments, a symbol X-Y is used as an abbreviation of each experiment. X represents the aging conditions: N-normal, DH-Dalvik Heap leak, NH-Native Heap leak. Y represents whether to clear recent task list or not. It has two options, in which C represents the clearance of recent task list and NC represents the holding of the recent task list. The details of the conducted experiments are described as follows. For the brief of the introduction, we only show the plan on the camera application.

1) Experiment N: Experiment N is designed under normal conditions. Since we observe that whether clearing the recent task or not will have impacts on the bug manifestation process, we further divide the experiment into two parts:

Experiment N-C: First open camera application, and then take 5 pictures. After that, press *back* button to the main menu, and then clear recent task. This **operation process** will be repeated automatically every 5 seconds and will last for two hours.

Experiment N-NC: Same with experiment N-C except that it removes the step of clear recent task.

2) Experiment DH: Experiment DH is designed for exploring the manifestation process of memory leak in Dalvik Heap. We first inject a memory leak in the camera application and then rebuild the Rom. Note that, in this experiment, we do not clear the recent task at the end of each process. The reason is that each time of clearing the recent task, the leaked memory will be reclaimed, and aging phenomenon will not show up. To clarify the point, we use the term Experiment DH-NC to denote the experiment if necessary.

3) Experiment NH: Experiment NH is designed for exploring the manifestation process of memory leak in Native Heap. We further divide it into two parts: Experiment NH-CP and NH-PP according to the priorities of processes (cached process and persistent process).

Experiment NH-CP: First install *Insert-native-memory.apk* on the phone and inject a Native Heap leak with the rate 200k/s, then press the *back* button to create a **c**ached **p**rocess (CP). Except for this, the procedure of the experiment is same as Experiment N.

Experiment NH-PP: Firstly, set Insert-native-memory.apk as a **p**ersistent **p**rocess(PP) . Secondly, we inject a Native Heap leak of 200k/s, and then press the *back* button. Except for these, the procedure of the experiment is same as Experiment N.

Note that, in this experiment, we clear the recent task at the end of each process to get rid of the risk of software aging caused by memory leak in Dalvik Heap. To clarify the point, we use the terms NH-CP-C and NH-PP-C as alternative terms for NH-CP and NH-PP if necessary.

III. Experimental results

In this section, we present and analyze the experimental results to answer the three above proposed research questions.

A. Answer RQ1

To answer RQ1, we make a comparison of the response time of an application (camera or calculator in this work) under different conditions.

1) Comparison of N-NC and DH-NC: Fig.2 shows the trend of each operation process's response time with the proceeding of Experiment N-C and DH-NC. Fig.2(a)-2(d) presents the results for the camera application and calculator application respectively, under the conditions of whether injecting a memory leak in Dalvik Heap or not. It can be observed from Fig.2(b) that due to the existence of memory leak in Dalvik Heap, the response time of camera may have big increasing intermittently, which will influence the user perception.

The phenomenon can also be observed in Fig.2(d), which shows even greater fluctuation than Fig.2(b). For the calculator application, sometimes it will take much longer time to open the application comparing with the previous operation processes. With the proceeding of the experiment, finally the

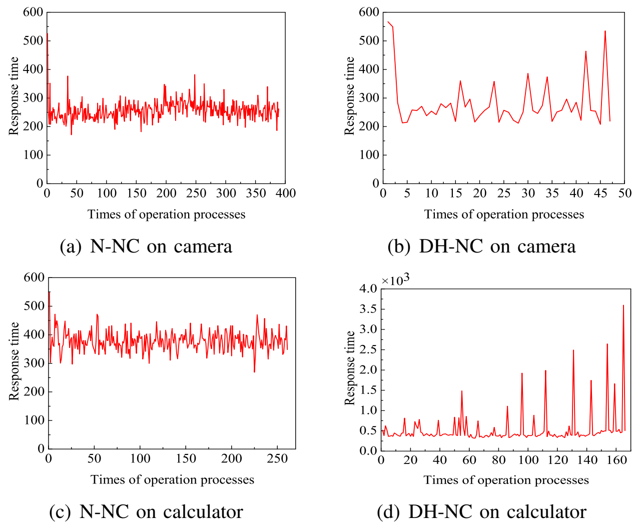


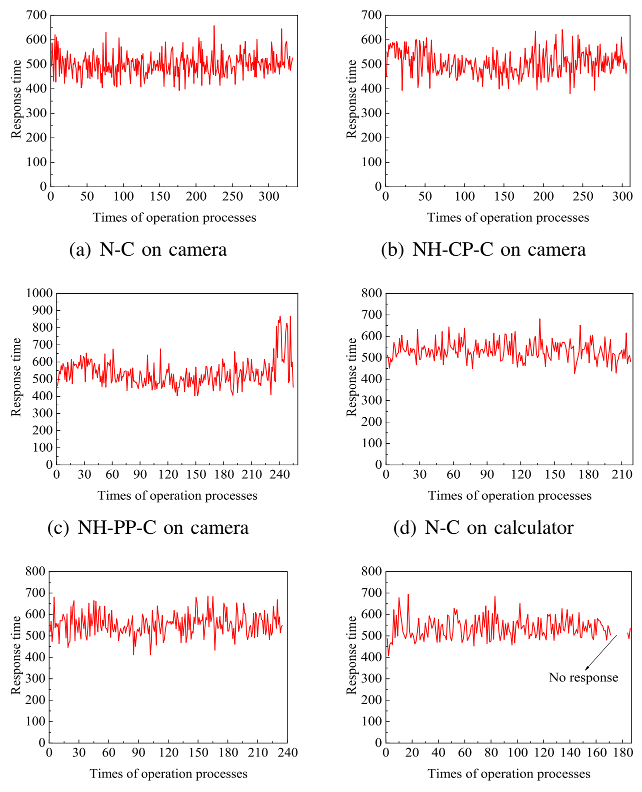
Fig. 2: Trends of the response time in Experiments N-NC and DH-NC

interface of calculator will not response for users' actions, and a failure message will be presented on the screen to show the crash of the application. After tracing the failure, we observe that the suddenly increasing of response time is mainly caused by the suspend of other threads during the time of GC working. It is occurred after *onResume()* method is called.

From above discussions, we can conclude that memory leak in Dalvik Heap influences user perception from two aspects: (1) an increasing trend in general and frequently nonnegligible fluctuations in local area of the response time as time goes by; (2) a crash of an application after long-time running.

2) Comparison of N-C, NH-CP-C, and NH-PP-C: Fig.3 shows the trend of each operation process's response time with the proceeding of Experiments N-C, NH-CP-C and NH-PP-C. Fig.4 shows the time to complete each operation process of Experiment NH-PP-C, in which x-axis represents the indices of operation processes and y-axis represents the completion time with the unit of a second. As shown in Fig.3(a) and 3(b), the response time of the camera application has almost no noticeable difference in Experiment NH-CP-C comparing with the N-C one, while no distinct difference and abnormal points can be found in both experiments. It implies that the Native Heap leak occurring in cached process have no obvious influence on the user perception.

Comparatively, memory leak in Native Heap persistent process shows a distinguishable phenomenon for users as shown in Fig.3(c). After a long-time running (e.g. after 234^{th} operation process in our experiment), the launching time of the camera application suddenly increases and starts to have great fluctuations, which means much longer time is necessary for users to open the application. From Fig.4, it can be observed that in the last 8 minutes of the experiment, not only the launching time of camera increases, but also unsmooth are detected when taking pictures, and the system responds users'



(a) N-C on camera (b) NH-CP-C on camera (c) NH-PP-C on camera (d) N-C on calculator (e) NH-CP-C on calculator (f) NH-PP-C on calculator

Fig. 3: Trends of the response time in Experiments N-C, NH-CP-C, NH-PP-C

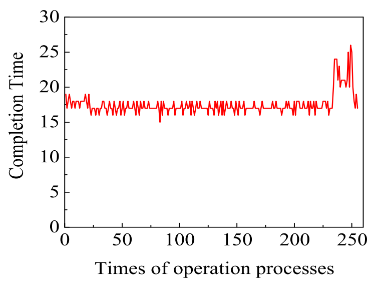
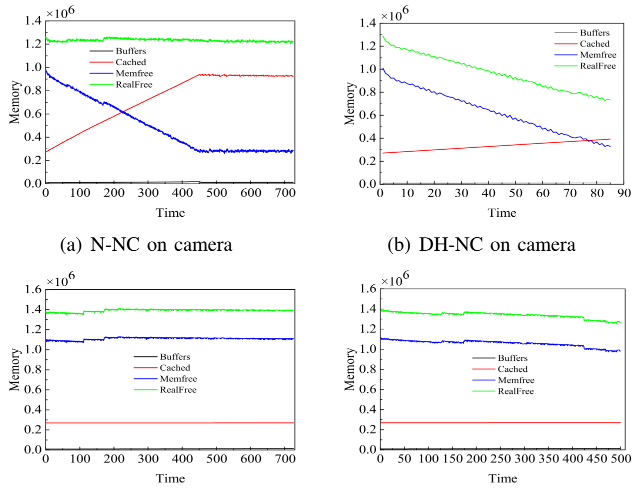


Fig. 4: The trend of completion time of operation processes in Experiment NH-PP-C on the camera application

actions slower than before. Although camera will work as normal as Experiment N-C due to the working of LMK after a long time, all these phenomena imply that Native Heap leak in persistent process will make the system unstable and have a great influence on the user perception.

As for the calculator application, similar phenomenon is observed in Experiment NH-CP-C and Experiment NH-PP-C, i.e., first the application works normally; then it starts to be abnormal after a number of repetitions of the operation processes and will take a period of time; finally it returns to the normal behavior. The difference of the two experiments is



(a) N-NC on camera (b) DH-NC on camera (c) N-NC on calculator (d) DH-NC on calculator

Fig. 5: Trends of the system memory factors in Experiment N-NC and DH-NC

mainly on the second phase, which will take about 6 minutes in our experiment. In this phase, the mobile phone has no response for users' actions, so there is no record of response time during this phase as seen in Fig.3(f). The reason is that, as *trebuchet* (desktop process of Cyanogenmod) owns a lower priority than persistent process, it is killed and restarted frequently by LMK due to the memory leak, making the system have no "energy" to response users' actions.

From the above discussions, we can conclude that memory leak in Native Heap will influence user perception from two aspects: (1) Native Heap leak in a cached process has no significant influence from the user perception; (2) Native Heap leak in a persistent process will lead to an increasing of the response time for users' actions after long-time running, and this phenomenon will last for a long time.

B. *Answer RQ2*

To answer RQ2, we compare the trends of memory information consuming by the whole system and an application (camera or calculator in this work) under different conditions. We design the experiments on two data types related with memory, i.e., memory information for the whole system and specific memory information for an android process. For detailed information of the two types, please refer to the data collection part in Section 3.1.

1) Comparison of Experiments N-NC and DH-NC on the memory information for the whole system: Fig.5 shows the trend of each operation process's memory information with the proceeding of Experiments N-NC and DH-NC, in which the x-axis represents the processing time of experiment and y-axis shows the amount of memory with the unit of kilobyte. We can observe from Fig.5(a) that, for the normal condition without clearing recent task in camera, cached memory has a stable growth at first and then stay at a stable value around

930M. The free memory (i.e., Memfree in the figure) shows a decreasing trend until it reaches around 270M. As shown in Fig.5(a), although Memfree has a decreasing trend, no significant variance of RealFree (i.e., Buffers + Cached + Memfree) is observed in Experiment N-NC. After injecting memory leak in Dalvik Heap, MemFree and RealFree decreases rapidly as shown in Fig.5(b). This implies that Realfree but not Memfree can be employed in Android as an available memory indicator for software aging. Another noticeable phenomenon is that, until the experiment stops due to the crash of the application, there is still plenty of memory left in Android, for example around 730M for the camera application, and 1250M for calculator.

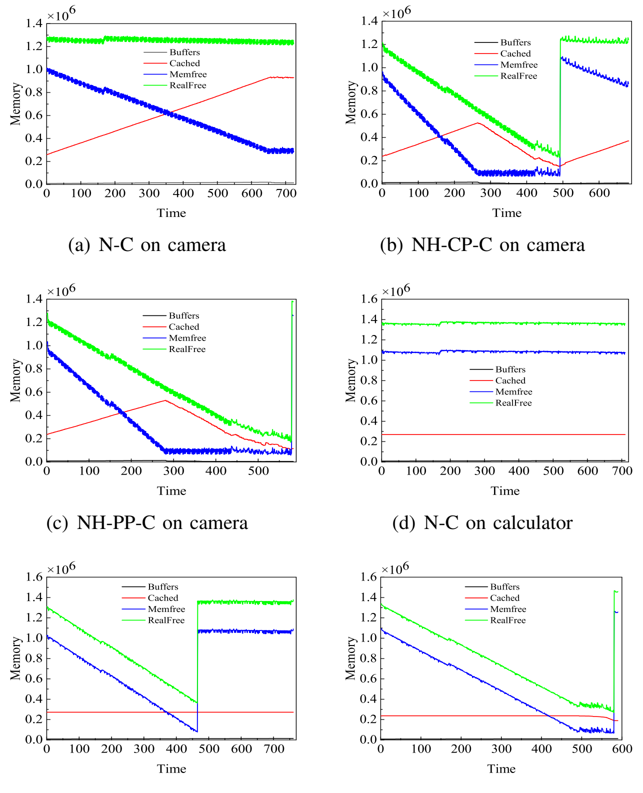
As shown in Fig.5(c) and Fig.5(d), similar phenomenon is observed in the experiment on calculator, except that cached memory keeps stable because the operation process of calculating does not take actions related with file read/write, so the RealFree and MemFree indicator have the same trend during the experiment.

In conclusion, from system's viewpoint, the decreasing of the indicator RealFree can indicate the existence of software aging. The case is similar with Linux [15], [16]. However, from the phenomena of the experiments, we can see that at the time of the failure occurring, there is still great amount of memory remaining. Thus, RealFree is not suitable to forecast a failure caused by memory leak in Dalvik Heap.

2) Comparison of N-C, NH-CP-C and NH-PP-C on the memory information for the whole system: Fig.6 shows the trend of each operation process's memory information with the proceeding of Experiments N-C, NH-CP-C and NH-PP-C. From Fig.6(b), we can observe that at first MemFree goes down rapidly to around 70M, and then it keeps steady. At the same time, Android system starts to reclaim cached memory, until it reaches the threshold (e.g. for the camera application, the threshold is 150M). After that, a suddenly increasing of MemFree arises. MemFree starts to be normal at the beginning of the next operation process, i.e., it shows the same status as Experiment N-C. Meanwhile, RealFree memory keeps a decreasing trend and finally works normally in this experiment. This phenomenon is caused by the LMK mechanism of Android. When a new memory allocation requirement arrives and there is inadequate free memory left for it, LMK starts to work and checks the priority level of Android processes currently running in the system. Processes owing lower priority will be killed by LMK. For example, in this experiment the cached process containing the Native Heap leak is killed due to its low priority.

As illustrated in Fig.6(c), the phenomenon of Native Heap leak in the persistent process is similar with that in Experiment NH-CP-C, i.e., LMK starts to kill processes when free memory is low. The only difference is that the minimum of RealFree is lower than that in Experiment NH-PP-C due to the high priority of a persistent process.

As shown in Fig.6(d)-6(f), the phenomenon observed in calculator has a little difference comparing with that for the camera application. The cached memory keeps at a steady



(a) N-C on camera
(b) NH-CP-C on camera
(c) NH-PP-C on camera
(d) N-C on calculator
(e) NH-CP-C on calculator
(f) NH-PP-C on calculator

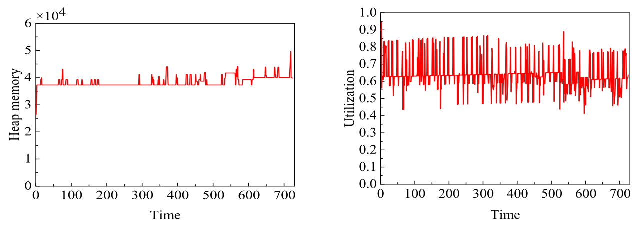
Fig. 6: Trends of the system memory factors in Experiments N-C, NH-CP-C and NH-PP-C

value even when free memory is in a low condition. So the trends of RealFree and MemFree are similar as that of the RealFree in the experiment on camera.

Note that, swap space has been used as an indicator of software aging in Linux [17], [6], [18], [19]. However, from the experimental results, we can see the value of used swap space keeps zero all the time. Thus, the indicator cannot be used in Android system. In Linux, some space of disk is used as swap space when the amount of physical memory is full, and inactive pages in memory will be moved to swap space. Differently, Android does not offer swap space for memory in default.

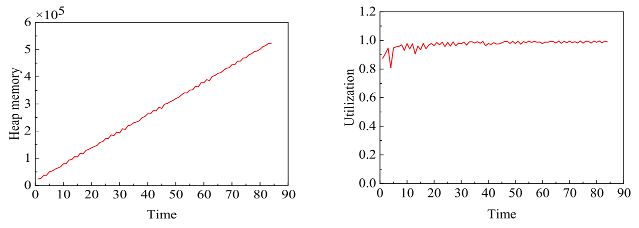
From the above discussions, we can conclude that from system's viewpoint: (1) although memory leak in Native Heap will lead to a decreasing trend of MemFree and RealFree and make little available memory left at last, Android can self-recover from that. This phenomenon makes the traditional prediction approaches utilized in Linux, i.e., using free memory as an indicator, cannot be applied in Android; (2) swap space cannot be taken as an indicator of software aging in Android.

3) Comparison of Experiments N-NC and DH-NC on specific memory information for an android process: Fig.7-10 shows the trends of Android specific memory information occupied by a process with the proceeding of Experiment N-



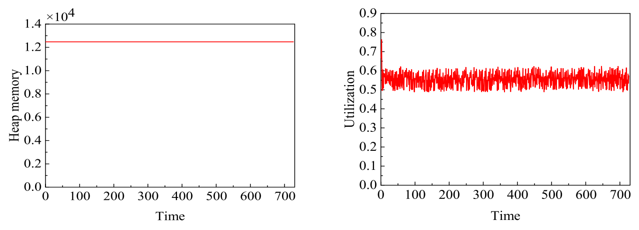
(a) Dalvik Heap Size (b) Dalvik Heap Utilization

Fig. 7: Trends of specific memory factor for an android process in Experiment N-NC on the camera application



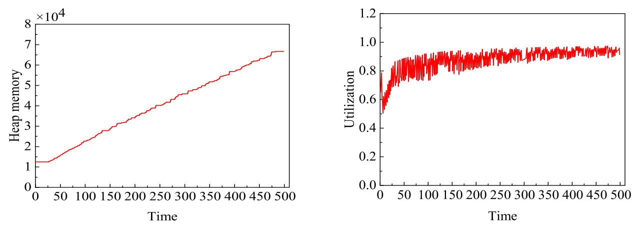
(a) Dalvik Heap Size (b) Dalvik Heap Utilization

Fig. 8: Trends of specific memory factor for an android process in Experiment DH-NC on the camera application



(a) Dalvik Heap Size (b) Dalvik Heap Utilization

Fig. 9: Trends of specific memory factor for an android process in Experiment N-NC on the calculator application



(a) Dalvik Heap Size (b) Dalvik Heap Utilization

Fig. 10: Trends of specific memory factor for an android process in Experiment DH-NC on the calculator application

NC and DH-NC, but not N-C and DH-C. The Dalvik Heap size and the heap's utilization are only analyzed in N-NC and DH-NC because once a user clears the application from recent task list, the process will be killed and the leaked memory be reclaimed. In the figures, x-axis represents the processing time of an experiment. Y-axis of Fig.7(a)-10(a) means the total allocated Dalvik Heap memory of the process with the unit KB, and y-axis of Fig.7(b)-10(b) represents the used percentage of Dalvik Heap. During the processing of experiments without memory leak, Dalvik Heap size stays stable as shown in Fig.7(a) and Fig.9(a). As illustrated in Fig.7(b) and Fig.9(b), the utilized percentage of Dalvik Heap for the camera application in normal conditions is between 0.4 and 0.9, while for the calculator application the value is between 0.5 and 0.7. The reason for the fluctuations is that after a user presses the *back* button, the heap memory occupied by the activity will be reclaimed, which leads to a suddenly decreasing of the heap usage.

Memory leak in Dalvik Heap will cause an increasing trend of Heap size, as shown in Fig.8(a) and Fig.10(a). However there is a noticeable difference between them. The allocated Heap size for the camera application can finally reach the maximal size of Dalvik Heap(i.e., 512M in this work) through dynamically increasing. The calculator application doesn't make use of largeHeap (a configuration that allows applications to use more Heap memory), so the maximal allocated size of Dalvik Heap is 192M by default. In this experiment, the calculator application crashes when the Heap Size reaches 67M, far from the maximal size of Dalvik Heap. The cases increase difficulties to predict the failure time.

As shown in Fig.8(b) and Fig.10(b), the heap utilization percentage for both applications grows rapidly and stays at a relatively higher value than the one without memory leak. The phenomenon shows that the utilization percentage of Dalvik Heap can be used as an indicator of memory leak in Dalvik Heap. Before the crash of the application, the percentage almost stays at high values which are close to each other. Thus, we cannot directly use the indicator to predict the failure or crash time of the application.

In conclusion, we observe that: (1) the allocated Dalvik Heap size keeps an increasing trend due to the memory leak, but the failure could occur before the allocated Dalvik Heap memory of a process raises to its maximal size; (2) the utilization percentage of Dalvik Heap shows an increasing trend due to the memory leak and then takes higher values than those in normal conditions.

IV. Conclusions and Future Work

In this paper, we have examined the various software aging manifestations in Android through injecting memory leaks in different heap areas and processes owing different priorities. The empirical results have been analyzed from both users' and system's viewpoints. Our future work will build rejuvenation models for aging processes in Android to obtain optimal rejuvenation strategies.

References

[1] https://code.google.com/p/android/.

[2] S. Yang, D. Yan, and A. Rountev, "Testing for poor responsiveness in android applications," in *Engineering of Mobile-Enabled Systems (MOBS),1st International Workshop on the*, pp. 1–6, IEEE, 2013.

[3] K. Nagata and S. Yamaguchi, "An android application launch analyzing system," in *Computing Technology and Information Management (ICCM), 8th International Conference on*, vol. 1, pp. 76–81, IEEE, 2012.

[4] T. Ongkosit and S. Takada, "Responsiveness analysis tool for android application," in *Proceedings of the 2nd International Workshop on Software Development Lifecycle for Mobile*, pp. 1–4, ACM, 2014.

[5] J. Araujo, V. Alves, D. Oliveira, P. Dias, B. Silva, and P. Maciel, "An investigative approach to software aging in android applications," in *Systems, Man, and Cybernetics (SMC), IEEE International Conference on*, pp. 1229–1234, IEEE, 2013.

[6] K. Vaidyanathan and K. S. Trivedi, "A measurement-based model for estimation of resource exhaustion in operational software systems," in *Software Reliability Engineering. Proceedings. 10th International Symposium on*, pp. 84–93, IEEE, 1999.

[7] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software aging analysis of the linux operating system," in *Software Reliability Engineering (ISSRE), IEEE 21st International Symposium on*, pp. 71–80, IEEE, 2010.

[8] Q. Ni, W. Sun, and S. Ma, "Memory leak detection in sun solaris os," in *Computer Science and Computational Technology. ISCSCT'08. International Symposium on*, vol. 2, pp. 703–707, IEEE, 2008.

[9] D. Robb, "Defragmenting really speeds up windows nt machines," *Spectrum, IEEE*, vol. 37, no. 9, pp. 74–77, 2000.

[10] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *Dependable Systems and Networks (DSN), IEEE/IFIP International Conference on*, pp. 447–456, IEEE, 2010.

[11] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo, "Memory leak analysis of mission-critical middleware," *Journal of Systems and Software*, vol. 83, no. 9, pp. 1556–1567, 2010.

[12] https://en.wikipedia.org/wiki/Android_(operating_system).

[13] https://developer.android.com/intl/zh-cn/tools/testing-support-library/index.html.

[14] https://developer.android.com/intl/zh-cn/tools/help/logcat.html.

[15] R. Matias, B. E. Costa, and A. Macedo, "Monitoring memory-related software aging: An exploratory study," in *Software Reliability Engineering Workshops (ISSREW), IEEE 23rd International Symposium on*, pp. 247–252, IEEE, 2012.

[16] F. Machida, A. Andrzejak, R. Matias, and E. Vicente, "On the effectiveness of mann-kendall test for detection of software aging," in *Software Reliability Engineering Workshops (ISSREW), IEEE 23rd International Symposium on*, pp. 269–274, IEEE, 2013.

[17] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi, "Analysis of software aging in a web server," *Reliability, IEEE Transactions on*, vol. 55, no. 3, pp. 411–420, 2006.

[18] K. Vaidyanathan and K. S. Trivedi, "A comprehensive model for software rejuvenation," *Dependable and Secure Computing, IEEE Transactions on*, vol. 2, no. 2, pp. 124–137, 2005.

[19] F. Machida, J. Xiang, K. Tadano, and Y. Maeno, "Software life-extension: a new countermeasure to software aging," in *Software Reliability Engineering Workshops (ISSREW), IEEE 23rd International Symposium on*, pp. 131–140, IEEE, 2012.