Stability Improvement of an Accelerated Android Operating System for Application Observation

Shoki Fukuda, Shun Kurihara
Electrical Engineering and Electronics, Kogakuin University Graduate School
Tokyo, Japan
cm16041@ns.kogakuin.ac.jp,
cm16015@ns.kogakuin.ac.jp

Masato Oguchi
Department of Information Sciences, Ochanomizu University
Tokyo, Japan
oguchi@is.ocha.ac.jp

Saneyasu Yamaguchi
Department of Information and Communications Engineering, Kogakuin University
Tokyo, Japan
sane@cc.kogakuin.ac.jp

Abstract— **Application analysis is important for smartphone markets and application developers. However, dynamic analysis involving the execution of the actual application is highly time consuming; thus, reducing the observation time is important issue. In this study, we focus on the Android operating system, which is based on a Linux kernel, and a method for accelerating the time flow speed as recognized by the processes in the system. The method accelerates the speed by modifying the implementation of time management in the kernel, thereby reducing the time required to monitor the behavior of applications. We discuss the stability and accuracy of the method with high acceleration ratios. First, we evaluated the method with a variety of ratios. The evaluation demonstrated that the method does not function accurately and stably with high acceleration ratios, whereas it is effective with low ratios. Second, we investigated the reason for this instability and showed that accelerated time flow causes several timers to experience timeouts, which results in system restarts. Third, we explored a method to prevent restarting, which we tested by disabling these timers and observing the system behavior.**

I. INTRODUCTION

The Android operating system has become one of the most popular platforms in mobile devices with a market share of 86.2 % in Q2 2016 [1]. This operating system supports a very large number of applications that are provided on application distributing sites by operating system developers [2], career services [3][4], and others.

Monitoring application behavior is important for various reasons, such as security purposes [5] or the estimation of power consumption [6]. Battery-draining applications can be identified effectively by monitoring the number of times WakeLock is acquired and Alarm is set [7][8]. In addition, some application distributing sites conduct application analyses as a service for users [5]. In addition, monitoring application behavior is an important process used by application developers to verify their own applications. Therefore, it is important to actually execute an application when assessing its behavior. However, analyses based on actual executions are highly time consuming [6][7][8]. Naturally, monitoring the behavior of an application over the course of one day takes 24 hours. We previously proposed a method to overcome this problem by constructing an accelerated environment [9][10][11], which was achieved by modifying the implementation of time flow management in the operating system kernel.

These studies demonstrated that this method was accurate for low acceleration ratios, such as twice the usual operating speed. However, the method has not been evaluated for high ratios, such as ten or one hundred times.

This paper introduces an accelerating method and presents an evaluation of its stability and accuracy with high acceleration ratios. We then describe the problems associated with stability-based evaluation. We explore the causes of system instability and explain our findings, namely that timers in the system are one of the important reasons. Finally, we describe the results we obtained after disabling these timers and investigating the effect of this solution on the system stability.

II. ANDROID APPLICATION

Android Application

Android applications are distributed by a variety of sites, such as Google Play Store [2], in which 2.4 million applications were registered in Sep. 2016 [12]. The huge number of applications makes it difficult for users to verify application behavior. Some application distributing sites perform application behavior checks as an important function of the service to their users [5]. However, dynamic application behavior analyses including practical application executions require a large amount of time as mentioned. Thus, conducting analyses of all the applications would be very difficult. This prompted us to investigate approaches to reduce the time required to dynamically analyze the behavior of applications as an issue of considerable importance.

Management of Time in Android

In this subsection, we explain time management in Android. This was explained in our previous work [9].

Operating systems using a Linux kernel, including Android, employ time management in their kernels. The processes in these systems obtain time-related information via system calls. The Linux kernel manages time based on the clock source provided by the hardware. The Android device used in this study has `gp_timer` and `dg_timer` as clock sources.

The time information presented by the clock source is stored in the variable `cycle_now`, and the difference from the last value is added to the time in the system. The time information is updated every tick, which is the interval of the periodical hardware timer interruption in Linux, in non-tickless kernels. Thus, the increase in `cycle_now` is added every tick. The tick is a parameter of the Linux kernel and is defined at compile time. In most kernels of Android operating systems, this is set to 10 ms. In contrast, tickles kernels do not use ticks as their update interval; instead, the amount by which the clock source has increased is added at the time of updating similar to a non-tickless kernel.

C. *Identification of Battery-Draining Applications with Dynamic Analyses*

Various methods for identifying battery-draining applications were previously proposed [6][7][8].

These studies focused on the behavior of the application in the screen-off state and discussed methods whereby applications with excessive battery consumption in this state could be identified. These studies argued that issuing WakeLocks to prevent the device from entering sleep mode, and setting Alarms, largely increased the power consumption. These researchers then proposed counting the number of times these issuing and setting operations were occurring in an attempt to identify battery-draining applications. Although these authors demonstrated that the above observations were effective for problem identification, they found that this approach consumed a significantly long time.

D. *Reducing Application Observation Time*

In our previous work, we proposed a method for constructing an environment for accelerated observation [9][10][11]. A detailed explanation is presented in section III.

First, we proposed to modify the implementation of time management in the Linux kernel to achieve acceleration [9]. The method accelerated the time flow recognized by processes in the system and reduced the time required to monitor the behavior of applications. This was achieved by improving speed at which `cycle_now` was increased. We constructed an accelerated Android operating system, and then demonstrated that the method could accelerate the time flow satisfactorily. The method [9] assumed that an application only ran on the device on which it was installed, i.e., a standalone environment, and the method supported only client-side Android devices. However, many of the recent applications communicate with server processes via the network. This limitation may present a serious problem. Second, we proposed to apply this method to both client-side applications and server-side services [8].

However, previously [9][10][11] we only evaluated the method with low acceleration ratios.

III. Reducing Application Observation Time by Modifying the Kernel

In this section, we introduce the implementation of our accelerating system [9].

Updating the time in a Linux kernel is executed by the function `timekeeping_get_ns()` in the file `time/timekeeping.c`. This function declares the `cycle_t` type variables named `cycle_now` and `cycle_delta`. Then, the time information is obtained from its clock source and is substituted for `cycle_now`, for which the increase since the value at the last call is calculated and the difference is added to *the time* recognized by the processes in the system. The time is referred to as the *wall clock*. We multiply `cycle_delta`, which indicates the amount of time that has passed since the last call, by increasing `cycle_now`, and this enables us to accelerate the time flow in the system.

Fig. 1 shows a sample of the modified source code, where `accel_cycle_now_delta` is the difference between the current time and the time of the last call, indicating the time that has passed since the last call. If acceleration is enabled, the rate at which *the time* is doubled is increased by increasing the value of `accel_cycle_now_delta`. Otherwise, the increase in *the time* is the same as usual.

The previous studies [9][10][11] compared the observation results obtained for environments with normal and low acceleration, and demonstrated that they were similar. That is, the accelerating method realized faster monitoring without a large decline in accuracy and without causing other problems, for example adversely affecting the stability. However, we did not previously evaluate high accelerating ratios.

IV. Evaluation

A. *Accuracy*

This subsection presents our investigation of the relation between the acceleration ratio and observation accuracy. The

```
char accel_enabled = 0;
unsigned long long accel_cycle_last=0;
unsigned long long accel_cycle_now_delta=0;
unsigned long long accel_cycle_mod_last=0;
static inline s64 timekeeping_get_ns(void)
{
    (omitted)
    cycle_now = clock->read(clock);
    accel_cycle_now_delta = cycle_now - accel_cycle_last;
    accel_cycle_last = cycle_now;
    if(accel_enabled){     //acceleration enabled
        cycle_now = accel_cycle_mod_last + accel_cycle_now_delta*2;
    }else{                 //acceleration disabled
        cycle_now = accel_cycle_mod_last + accel_cycle_now_delta;
    }
    accel_cycle_mod_last = cycle_now;
    cycle_delta = (cycle_now - clock->cycle_last) & clock->mask;
    (omitted)
}
```

Fig. 1. Modified source code

acceleration ratio was increased by 1, 2, 6, 24, 48, and 60 times to monitor the application behavior. The observation was performed by installing a set of applications, leaving the Android device untouched for one hour, and counting the number of times a WakeLock is acquired and the Alarm is set. Counting these events is important to identify battery-draining applications [7]. The application set was composed of the top 10 applications ranked in the *news & magazines* category of the Google Play Store on Oct. 24, 2015. The experimental environment is presented in Table I.

The experimental results are shown in Fig. 2 to Fig. 4. Figure 2 indicates that the results obtained with acceleration ratios less than or equal to six times are quite similar. This enabled us to conclude that a suitable acceleration, with which similar observation results can be obtained in shorter time, is achieved. The results we obtained for ratios larger than six times indicate that the normal and accelerated environments do not behave very similarly. Figure 3 and Fig. 4 show that the difference between the two environments increases as the acceleration ratio increases.

The maximum difference in the number of WakeLock acquisitions and Alarm sets is 26.6% and 14.9%, respectively. When the required accuracy is less than these values, an observation with an acceleration of 60 times is suitable. For example, finding the most battery-draining application by identifying the application that issues the largest number of WakeLocks, which was required previously [7], can be achieved by using an acceleration of 60 times.

TABLE I
EXPERIMENTAL ENVIRONMENT

Device Name	Nexus7 (2013)
OS	Android 5.0.1 (AOSP) with modified Linux kernel 3.4.0

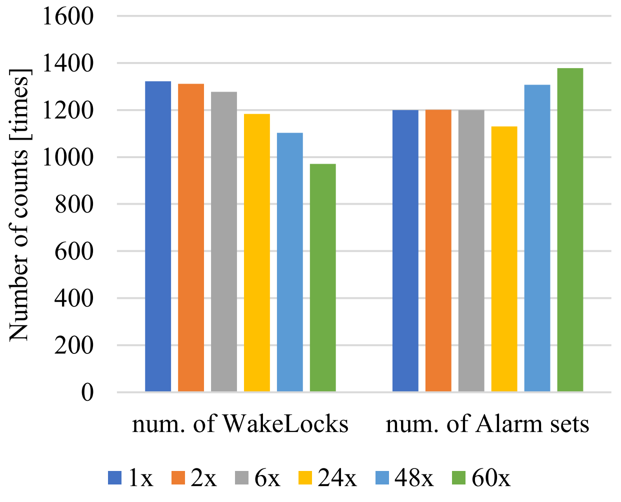

Fig. 2. Acceleration ratio and the number of WakeLocks and Alarm sets

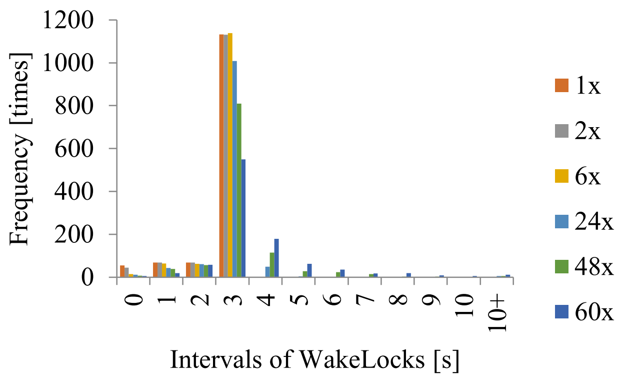

Fig. 3. Acceleration ratio and intervals of WakeLocks

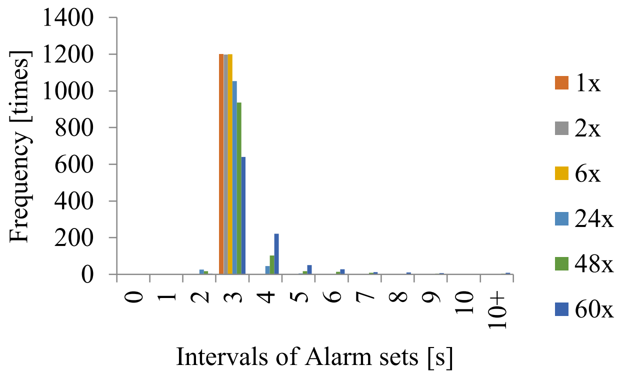

Fig. 4. Acceleration ratio and intervals of Alarm sets

B. *Stability*

In this subsection, we discuss the relation between the acceleration ratio and system stability. Our Android device remained untouched with the standard Home application displayed using a variety of acceleration ratios, after which the time the device was able to run stably was measured. For each evaluation, we rebooted the device and enabled acceleration 30 seconds after each boot. The experimental setup is presented in Table II.

Figure 5 shows the experimental results, with the vertical axis representing the system stability. This is the ratio for which the system succeeded in executing the Home application without system failure for 60 minutes of the operating system clock, i.e., one minute of real clock time accelerated 60 times. The results in the figure imply that the system could continue running the Home application at ratios of 60 times. In addition, we can see that the system stability declined as the ratio increased above 60 times.

All the system failures that occurred were reboots of the Android framework process, which runs in the user space. In all the cases, the Linux kernel neither froze nor rebooted. The command prompts connected via the Android Debug Bridge (ADB) continued running.

TABLE II
EXPERIMENTAL ENVIRONMENT

Device Name	Nexus7 (2013)
OS	Android 5.1.1 (AOSP) with modified Linux kernel 3.4.0

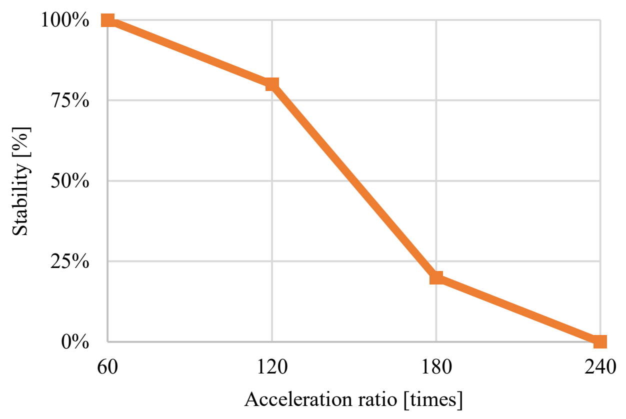

Fig. 5. Acceleration ratio and system stability

C. *Discussion*

In this subsection, we discuss the cause of the system failures attributed to reboots of the Android Framework processes. Figure 6 displays part of the information provided by the Android system log. The log indicates that detection of *Slow operation* and the timeout of some timers, such as the watchdog timer, caused process terminations. This clearly suggests that improving the speed of the time flow without enhancing the performance of the hardware devices results in the recognized consumed-time becoming longer. As a result, slow operation and timeouts were detected.

```
W/ActivityManager(11677): Slow operation: 1216ms so far, now
at startProcess: asking zygote to start proc
W/SurfaceFlinger(  179): Timed out waiting for hw vsync; faking it
W/WindowManager(11677): Window freeze timeout expired
I/Choreographer(11818): Skipped 43 frames!  The application may
be doing too much work on its main thread.
W/AudioFlinger(11357): power manager service died !!!
W/Watchdog(11677): *** WATCHDOG KILLING SYSTEM PROCESS:
Blocked in handler on ActivityManager (ActivityManager)
W/Watchdog(11677): *** GOODBYE!
I/lowmemorykiller(  176): Closing Activity Manager data
connection
E/Zygote  (11355): Exit zygote because system server (11677) has
terminated
```

Fig. 6. Part of the Android system log

D. *Effect of Disabling System Process Killing*

We subsequently disabled the killing system process caused by timeouts of the watchdog timers and *Activity Manager* in the Android operating system prior to evaluating the system stability. The experimental results are plotted in Fig. 7. The results in the figure imply that the system stability is improved by disabling these killing functions. Our experiments did not reveal a negative effect on the operating system as a result of disabling these functions. A detailed explanation of the modification on an Android implementation is provided in appendix A.

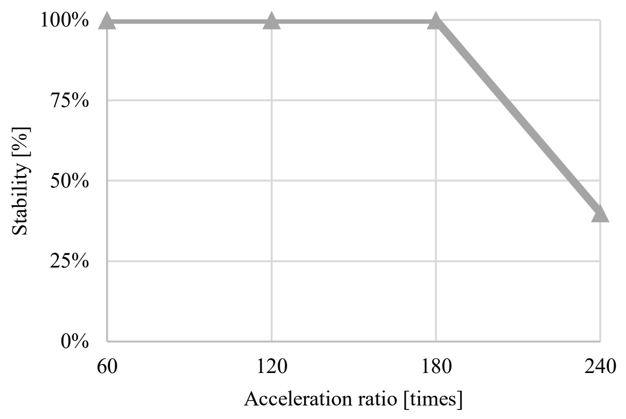

Fig. 7. Acceleration ratio and system stability

V. Related Work

This section introduces related work performed previously.

Enck et al. surveyed 1,100 popular Android applications when investigating security problems pertaining to smartphones [13]. They analyzed the applications with the decompiler and checked the field values. This work was based on a static analysis, and their discussion did not include a dynamic analysis.

The following work relates to a dynamic analysis. Methods for identifying battery-draining applications based on dynamic analyses were proposed [6][7][8]. These methods monitored of the number of times WakeLocks and Alarm sets were issued, and then estimated the power consumption of each application based on the monitoring results. However, these methods were problematic in that dynamic monitoring is highly time consuming and were unable to solve the problem. Petsas et al. proposed a method for analyzing applications, including dynamic analysis, for security purposes [14]. Their work involved applying the proposed method to actual malicious software and they presented the results of their analysis. However, they only proposed a method for analyzing the time and did not present a method for reducing the analyzing time. Yan et al. proposed a platform for analysis based on virtualization [15]. This platform also only provided a method for analyzing the time and did not contain a method for decreasing the time. We proposed a method for adjusting the performance and the power consumption based on dynamic monitoring [16]. This work did not mention a method for reducing monitoring time.

The following work involved time management in the kernel. Ferrari et al. evaluated the software-based IEEE 1588 implementation [17]. Their paper mentioned time-managing methods, including the use of the clock source, but did not propose a method for accelerating the time flow. Kobayashi et al. proposed a method for accelerating the time flow by modifying the kernel in order to reduce the time required to observe the system behavior [18]. Their work was pioneering and proposed to increment the value of `jiffies` to accelerate the time. However, this method cannot be used to accelerate the time of recent Linux kernels. In addition, the work provided no evaluation of the Android operating system.

Previously [9], we proposed a method for accelerating the time flow of the Android operating system. The current paper is based on the previous work. However, previously we did not evaluate the system stability. In addition, no method for improving the system stability was mentioned.

VI. CONCLUSION

In this paper, we introduced an important problem relating to the dynamic observation of Android applications, namely that this analysis is highly time consuming. We presented a method for reducing the observation time by modifying the implementation of time management in the kernel. We evaluated the method by using high acceleration ratios. Our experiments revealed the adverse effect of the system stability to be attributable to the system process rebooting in the user space. We then investigated the causes of system failure and found that process terminations caused by timeouts that occurred during accelerated time were an important cause. For testing purposes, we implemented the operating system such that these terminating functions were disabled, thereby improving the system stability, as our tests confirmed.

In future, we plan to evaluate our method with even higher accelerating ratios to enable us to propose a method for improving the system stability.

ACKNOWLEDGMENT

This work was supported by JST CREST Grant Number JPMJCR1503, Japan.

This work was supported by JSPS KAKENHI Grant Numbers 26730040, 15H02696, 17K00109.

REFERENCES

[1] IDC: Smartphone OS Market Share 2016, 2015, http://www.idc.com/prodserv/smartphone-os-market-share.jsp
[2] Google Play Store, https://play.google.com/store/apps
[3] NTT docomo d-market application&review, http://app.dcm-gate.com/
[4] au Market, http://market.kddi.com/update_info/
[5] Android and Security - Official Google Mobile Blog, http://googlemobile.blogspot.jp/2012/02/android-and-security.html
[6] Shun Kurihara, Shoki Fukuda, Shintaro Hamanaka, Masato Oguchi and Saneyasu Yamaguchi. “Application Power Consumption Estimation Considering Software Dependency in Android,” 2017 The International Conference on Ubiquitous Information Management and Communication (IMCOM2017), 2017
[7] Shun Kurihara, Shoki Fukuda, Ayano Koyanagi, Ayumu Kubota, Akihiro Nakarai, Masato Oguchi and Saneyasu Yamaguchi: “A Study on Identifying Battery-Draining Android Applications in Screen-Off State,” 2015 IEEE 4th Global Conference on Consumer Electronics (GCCE 2015), 2015.
[8] Shun Kurihara, Shoki Fukuda, Shintaro Hamanaka, Masato Oguchi and Saneyasu Yamaguchi. “Identifying Battery-Draining Applications by Monitoring Behavior in Screen-Off State in Android,” 2016 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW 2016)
[9] Shoki Fukuda, Shun Kurihara, Shintaro Hamanaka, Masato Oguchi and Saneyasu Yamaguchi: “Accelerated Application Monitoring Environment of Android,” 2016 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW), 2016
[10] Shoki Fukuda, Shun Kurihara, Shintaro Hamanaka, Masato Oguchi and Saneyasu Yamaguchi: “An Accelerated Application Monitoring Environment with Accelerated Servers,” 2016 IEEE The 5th IEEE Global Conference on Consumer Electronics (GCCE 2016), 2016
[11] Shoki Fukuda, Shun Kurihara, Shintaro Hamanaka, Masato Oguchi and Saneyasu Yamaguchi. “Accelerated Test for Applications with Client Application and Server Software,” 2017 The International Conference on Ubiquitous Information Management and Communication (IMCOM2017), 2017
[12] Number of Google Play Store apps 2015 Statistic, http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store
[13] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A study of android application security. In Proceedings of the 20th USENIX conference on Security (SEC'11). USENIX Association, Berkeley, CA, USA, 21-21.
[14] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos,Michalis Polychronakis, Sotiris Ioannidis, “Rage against the virtual machine: hindering dynamic analysis of Android malware,” EuroSec '14 Proceedings of the Seventh European Workshop on System Security, Article No. 5
[15] Lok Kwong Yan, Heng Yin, “DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis,” 21st USENIX Security Symposium, 2012
[16] K. Nagata, S. Yamaguchi and H. Ogawa, "A Power Saving Method with Consideration of Performance in Android Terminals," 2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing, Fukuoka, 2012, pp. 578-585. doi: 10.1109/UIC-ATC.2012.133
[17] P. Ferrari, A. Flammini, S. Rinaldi, A. Bondavalli and F. Brancati, “Evaluation of timestamping uncertainty in a software-based IEEE1588 implementation,” Instrumentation and Measurement Technology Conference (I2MTC), 2011 IEEE, Binjiang, 2011, pp. 1-6.
[18] Yoshitake Kobayashi, “Linux Kernel Acceleration for Long-term Testing,” CELF Embedded Linux Conference Europe 2010.

APPENDIX

Disabling Process Terminating Function

In section IV. C, we disabled the functions of process termination for testing. In this appendix, we explain the details of the implementation. We disabled the terminating functions in the Activity Manager and Watchdog timers. The former is implemented in `frameworks/base/services/core/java/com/android/server/am/ProcessRecord.java` as shown in Fig. 8. The latter is implemented in `frameworks/base/services/core/java/com/android/server/Watchdog.java` as in Fig. 9. Lines 529 to 534 in Fig. 8 were converted into comments to disable the termination. Similarly, in the code in Fig. 9, lines 477 to 478 were changed to comments.

```
(omitted)
    void kill(String reason, boolean noisy) {
        if (!killedByAm) {
            if (noisy) {
                Slog.i(ActivityManagerService.TAG, "Killing " + toShortString() + " (adj " + setAdj
                        + "): " + reason);
            }
            EventLog.writeEvent(EventLogTags.AM_KILL, userId, pid, processName, setAdj, reason);
            Process.killProcessQuiet(pid);
            Process.killProcessGroup(info.uid, pid);
            if (!persistent) {
                killed = true;
                killedByAm = true;
            }
        }
    }
(omitted)
```

Fig. 8. Process kill function in Activity Manager

```
(omitted)
            if (debuggerWasConnected >= 2) {
                Slog.w(TAG, "Debugger connected: Watchdog is *not* killing the system process");
            } else if (debuggerWasConnected > 0) {
                Slog.w(TAG, "Debugger was connected: Watchdog is *not* killing the system process");
            } else if (!allowRestart) {
                Slog.w(TAG, "Restart not allowed: Watchdog is *not* killing the system process");
            } else {
                Slog.w(TAG, "*** WATCHDOG KILLING SYSTEM PROCESS: " + subject);
                for (int i=0; i<blockedCheckers.size(); i++) {
                    Slog.w(TAG, blockedCheckers.get(i).getName() + " stack trace:");
                    StackTraceElement[] stackTrace
                            = blockedCheckers.get(i).getThread().getStackTrace();
                    for (StackTraceElement element: stackTrace) {
                        Slog.w(TAG, "    at " + element);
                    }
                }
                Slog.w(TAG, "*** GOODBYE!");
                Process.killProcess(Process.myPid());
                System.exit(10);
            }
(omitted)
```

Fig. 9. Process kill function in Watch Dog timer