Control Theoretic Adaptive Monitoring Tools for the Android Platform

DAVID REYNOLDS
Department of Computer Science
Texas State University
San Marcos, USA
dr1299@txstate.edu

MINA GUIRGUIS
Department of Computer Science
Texas State University
San Marcos, USA
msg@txstate.edu

Abstract— **With the escalation of attacks that target mobile devices there is an increasing need for efficient monitoring tools. Due to computation, storage, and power constraints of the devices, these monitoring tools may overload the entire system causing severe performance degradation for the running applications. Moreover, they can be pushed by resource-intensive applications and may not perform an adequate monitoring job. To that end, in this project we develop adaptive monitoring tools that are capable of monitoring a wide range of information while efficiently utilizing the available resources on the mobile device. We apply control theoretic techniques to design monitoring functionalities, such as process, integrity and network monitors, that utilize the underlying resources efficiently. This is achieved through controllers that dynamically select the duty cycles of the monitors. Our models and results are validated on the Android platform through emulation and real implementation on the Nexus 7 tablet.**

I. INTRODUCTION

Motivation: The Android platform is becoming increasingly popular due to its open-source framework. With more than 400 million activated Android devices, it is being considered by various government organizations, including DoD, in diverse settings. Many users utilize Android devices in their daily work routines such as: checking email, logging into different servers, and accessing on-line services. With the rise in popularity of Android devices there has also been a significant rise, as much as 400%, in number of attacks on mobile devices [1]. With intensifying attacks that seek to steal private data, track keystrokes, and increase the phone bill by texting, calling or generating traffic, such organizations become stakeholders in the security of the Android platform. While an Android device does have security measures embedded within the system, there are still major vulnerabilities to protect against. This rise in malicious activity increases the need for the deployment of efficient monitoring tools. Due to computation, storage and power constraints of the mobile devices, however, these tools need to adjust their resource consumption in tandem with the applications running. This is achieved through the use and application of control theory to develop monitoring tools.

In this project we present DriodMonitor - a monitoring application - that utilizes control theoretic controllers to monitor the device while limiting its own use of system resources. DroidMonitor can be easily extended by including additional monitors to gather data that is requested by the user. The user sets a target utilization value of the resources that DroidMonitor needs to match. This implies that DroidMonitor will actively fight to obtain the allowed resources when the system is under heavy use, and will also limit its resource consumption when the device is in a period of light use. Thus giving the user the ability to choose DroidMonitor's effect on the system as a whole and adjust the amount of data gathered by the monitors within.

Throughout this project, we have experimented with different types of controllers including the traditional control theoretic ones such as the Proportional (P), Proportional Integral (PI) and Proportional Integral Derivative (PID). Many experiments were conducted in order to assess their performance in terms of their stability and response to various stimuli in controlling DroidMonitor's own resource consumption. Our models and results are validated on the Android platform through emulation and real implementation on the Nexus 7 tablet.

Regarding our implementation of DroidMonitor, it is important to mention that while the core abilities of DriodMonitor do not rely on the Android device being rooted, it is required for the network monitor portion of this tool. It is our opinion that rooting a device is more of a risk than it is worth. By rooting a device the Android device presents a much easier target for malicious attacks, while admittedly this would only increase the need for monitoring the device, rooting the device is counter-intuitive to the overall security of the Android operating system. This does place limits on what can be monitored from the application level, namely network traffic sniffing. In order to implement the much sought after mobile network monitoring aspect of DroidMonitor, the testing devices have been rooted.

Paper organization: This report is organized as follows: In Section 2 we discuss the problem in more details elaborating on pervious related work in the area. Section 3 describes our methodology and the application of control theoretic techniques in DriodMonitor. We report on our results in Section 4 and conclude the report in Section 5 with a summary.

II. RELATED WORK

There has been some previous work accomplished in order to address this issue. There are many Android applications that can be easily downloaded and installed from Google Play that will monitor the active processes of the device. Popular applications that fall into this category include OS Monitor [2], Android Assistant [3], and Advanced Task Killer [4]. These applications often fall short in many areas that need to be addressed. One of the main areas in which these applications fail is, they do not continue monitoring the device when not running in the foreground. The user is not able to monitor the activity of an application while it is running in the foreground since the monitoring application has to be in the foreground in order to monitor the running processes. Despite only monitoring the device when in the foreground monitoring applications also often use large amounts of the device's limited resources, while failing to maintain any record of past resource consumption. In order to obtain a clear picture of what the device is being used for, the monitoring device should be as unobtrusive on the system as possible while gathering the needed granularity of data for the information to be useful to the user.

Google has taken a proactive approach in an attempt to limit the number of malicious content that is released. All applications looking to be published on Google Play must first be checked by Bouncer, a program designed to catch malicious applications. However, Bouncer is easily bypassed, and does nothing to prevent third party sites from releasing malicious content. There is a strong need to be able to compare applications post distribution, according to the malware genome project 86% of malicious content on Android devices are "repackaged versions of legitimate applications with malicious payloads." [5] There has been research in the area of detecting repackaged applications on third party websites, DroidMOSS [6] uses fuzzy hashing to detect repackaged applications being hosted on alternative application stores.

A wide range of research has been accomplished regarding Android security. Crowdroid [7] gathers system calls made on the Android device, then sends the data to a remote server to be analysed by a behavior-based malware detection remote server. TaintDroid [8] tracks an applications use of the users private data in an attempt to detect inappropriate use of that data. Kirin security service performs lightweight certification of applications to mitigate malware at install time [9].

III. The General Framework

In this section we outline the general framework for DroidMonitor. We describe the different controllers that are based on feedback control systems.

A. The Architecture

DroidMonitor is an extensible framework that is composed of: (1) a set of monitors, (2) a feedback monitor and (3) controllers. Each monitor is responsible for performing a specific task such as acquiring CPU utilization of the running processes, hashing application binaries and sending the results to the cloud, or sniffing network traffic. DroidMonitor is designed to allow the addition and removal of monitors to be quick and easy. The controller decides the duty cycles of the monitors to ensure that DroidMonitor does not use too few or too many resources. This is achieved by allowing DroidMonitor to sleep for a period of time that is dynamically chosen based on the load of the system and based on the resources used by DriodMonitor. The feedback monitor reports the resource utilization of DriodMonitor itself back to the controller to decide future actions. Figure III-A shows a block diagram of the proposed architecture.

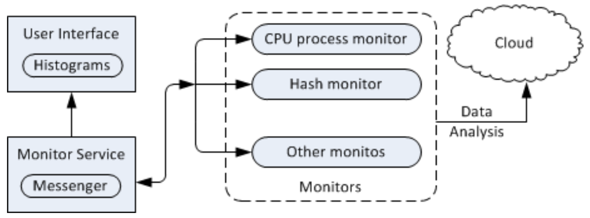

Fig. 1. The DroidMonitor Framework

B. The Monitors

Each monitor is run in its own thread that is controlled by the Monitor Service, and largely by the user through the user interface. The current version of DroidMonitor includes three monitors: a monitor for recording the CPU utilization of each of the running processes, a monitor for hashing the application binaries for integrity checks and another one for sniffing network packets through TCPdump [10].

C. The Feedback Monitor

The feedback monitor is run in an independent thread from any of the monitors, an OFF period is determined based on the current resource load and the current control algorithm being used. The CPU consumption of DroidMonitor is obtained through one of the monitors. Then, the controller uses this information, along with information detailing the set target consumption, to set an OFF period for DroidMonitor. The OFF period is passed to each of the monitors via the Monitor Messenger and then each monitor will adjust their sleep cycle to match the set OFF period. If DroidMonitor is using more than its allowed resources it will increase the time spent in a wait state, and vice versa if the monitor is not using all of its allowed resources. By increasing its requested resources from the system it can actively push back on other resource heavy programs in order to match the user set resource constraints. This allows the user to set the granularity of data gathered, and prevent other applications from monopolizing the devices resources. The feedback monitor is an integral process in the Feedback Loop and can be seen in Figure 2.

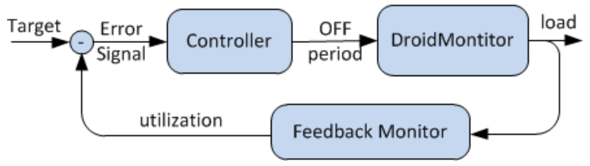

Fig. 2. The Feedback Loop

D. The Controllers

DroidMonitor implements different types of controllers to adjust its own resource utilization at a target value T that is chosen by the user. The following controllers are available:

- **Ratio-based Controller:** In this controller, DriodMonitor calculates its P_{on} period, the time taken to execute all the monitors, and uses Equation 1 to adjust its P_{off} period.

$$P_{off} = P_{on} \times (\frac{1}{T} - 1) \tag{1}$$

- **Proportional Integral Controller:** In this controller, DroidMonitor adjusts its P_{off} based on its previous value and the error signal at time, t. The error signal, e, is the utilization of DroidMonitor, ρ_{DM}, minus its target value T. K is the gain of the controller that adjusts its aggressiveness. Larger values of K cause the controller to react swiftly to small variations, while smaller values makes the controller more stable and less reactive. When the utilization of DroidMonitor is at its target value, the error signal would be zero and the P_{off} period would not change. Equation 2 describes the PI controller.

$$P_{off} = P_{off} + K \times e \tag{2}$$

- **Proportional Integral Differential Controller:** In this controller, DroidMonitor adapts P_{off} using a controller based on a standard PID controller. The user can adjust each individual value of K at the user interface. Equation 3 describes how the P_{off} period is selected with a PID controller.

$$P_{off} = K_p e(t) + K_i \int_0^t e(t) + K_d \frac{d}{dt} e(t) \tag{3}$$

IV. Performance Evaluation

In this section, we describe our evaluation of DroidMonitor on the Android Virtual Devices (AVD) and on the Nexus 7 tablet.

A. The Setup

We thoroughly tested the implemented control algorithms using an AVD running Google API level 15 with WVGA800 skin. Each Test had three test monitors included within DroidMonitor.

The first monitor is a Process Monitor, this monitor gathers and logs information on all running applications on the device. The information is gathered from the proc/[PID]/ and written to the device in a secure file that can only be written to and read from DroidMonitor. This file, however, can be pulled from the device and analyzed by the user. This Process Monitor also feeds information about DroidMonitor itself back to the feedback monitor.

The second monitor is named Hash Monitor. This monitor computes an MD5 hash of every installed application on the phone. Hash Monitor was created mainly to test two different ideas. The first idea was to see how the controllers would react to a sudden burst in activity from its own process when MD5 hashes are computed, and the second to compare identical applications installed on separate but similar devices. The comparison can be used to identify applications claiming to be legitimate applications that have been tampered with.

The last monitor is a Network Monitor. This monitor launches and collects packets captured by tcpdump [10]. These packets are then written to a secure file similar to that used by the Process Monitor. The resource consumption of tcpdump is added to the total resource consumption of DroidMonitor in order to keep DroidMonitor's total resource consumption close to the target set by the user. We report below on the results obtained from our experiments with the implemented controllers.

B. Results

Ratio-based Controller

This simple and straight forward controller does a decent job at maintaining the target resource usage. However, since the P_{off} are dependent on the P_{on} periods of the current running monitors the actual resource consumption has a higher deviation than other controllers. This is due to uncontrollable variation in P_{on}, and the inability to obtain a complete on period for all of the DroidMonitor's threads. While the difference is slight and easily adjusted for, the main downfall of the ratio-based controller is its response to other heavy resource consuming processes. This controller lacks the ability to push back against these processes and drops drastically below target consumption failing to gather the needed data. The behavior of this controller can be seen in Figure 3.

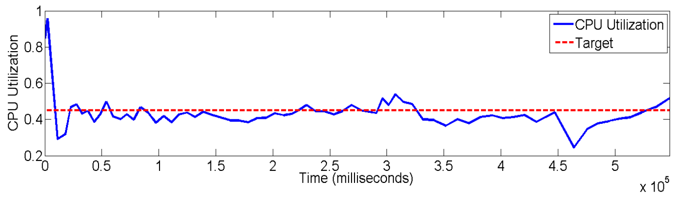

Fig. 3. P Controller: T = 45

Proportional Integral Controller

Our implementation of the PI controller uses the parameter K that decides the aggressiveness of the controller in its reaction to the error signal. Figure 4 shows the utilization of DroidMonitor when the target CPU usage was set to 30% under a choice of K of 100. While it takes a considerable amount of time to reach the target, once it is reached the choice of K of 100 seems to keep the utilization close to the target. In Figure 5, K is set to 1000. As one can see the target is reached at a much faster pace than when K was set to 100, and remains closer to the target value. Finally, in Figure 6 it is possible to see an extreme example of an aggressive controller with K set to 10000 where the target is reached by a drastic overshot and then oscillates wildly around the target area.

As we can see from the previous three graphs it appears that when K is set to a value of, or near 1000, the controller achieves the proper balance between swiftly reaching the target

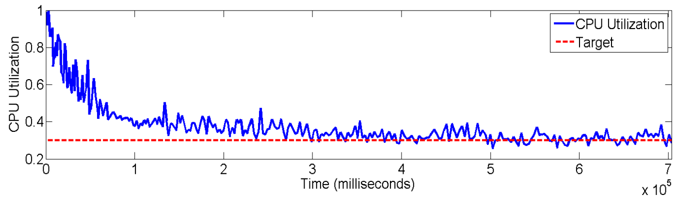

Fig. 4. PI Controller: T = 30, K = 100

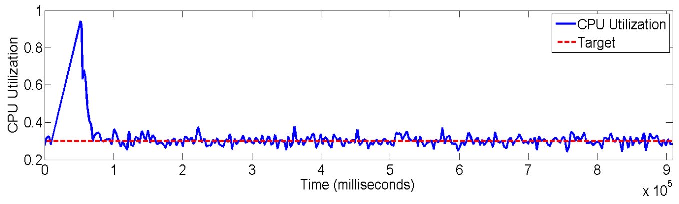

Fig. 5. PI Controller: T = 30, K = 1000

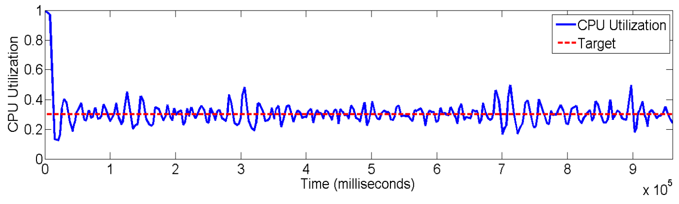

Fig. 6. PI Controller: T = 30, K = 10000

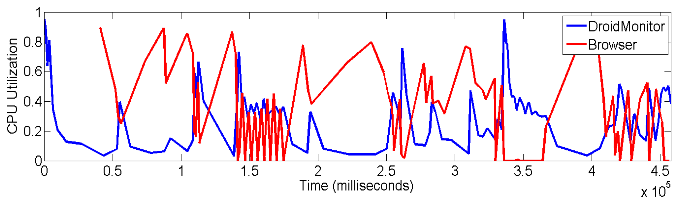

Fig. 7. PI Controller: T = 30, K = 1000, Browsing the Internet

and staying near the target. However, as seen in Figure 7 - where we introduce a browsing activity - a choice of a K value of 1000 does not respond well when fighting for resources. The high K value is too reactive to remain at or near the target value during a common activity such as browsing the Internet on the device. When K is set to a smaller value, such as 100, and the experiment is repeated it becomes clear that a less reactive value for K is needed when the device is under normal use. Figure 8 shows the behavior with K set to 100 with browsing activity.

The results above suggest that it would be worthwhile to adjust the value of K dynamically to affect the behavior of the controller. Figure 9 and Figure 10 show the results of a PI controller where the value of K is adjusted linearly between 100 and 1000 based on the overall utilization of the CPU. In particular, when the device is under low usage, the value of K is increased appropriately to swiftly reach and maintain the target resource consumption. When the device is experiencing heavy use and spikes in resource consumption the value of K is lowered to prevent jittery and unpredictable behavior from the monitors. These two figures reveal that not only does this insure that DroidMonitor can quickly reach target resource consumption, but can also remain at target resource consumption when other applications are trying to use the same resources.

Proportional Integral Differential Controller

Figure 11 shows the behavior of the PID controller with browsing activity. K_P, K_I and K_D were set to 1, 1, and 10 respectively. Originally the PID controller was expected to be the best possible controller for this device. However, after many experiments it became clear that this controller often failed to meet DroidMonitor's needs. Only one set of variables was able to quickly approach the target consumptions, yet as can be seen by Figure 11, the PID controller is not able to maintain the target resource use while other activities are being run on the device. We believe that this is primarily due to the selection of the weights for the components of the controller and tuning its performance. We plan to investigate this issue further in the future.

C. DroidMonitor on Nexus 7

Along with ample testing completed on an Android Virtual Device, DroidMonitor was also tested on a Nexus 7. The 16 GB Nexus 7 runs Android version 4.2.1 (Jelly Bean) on a NVIDIA(R) Tegra(R) 3 quad-core processor. Figure 12 shows the DroidMonitor interface.

The Nexus 7 offered a drastically different environment to test on when compared to the virtual device. We found that each monitor completed its task faster, and at much less resource cost to the system. While running DroidMonitor on a virtual device the CPU monitor could be run with no controller and use as much as 75% of the CPU, when the same test was completed on the Nexus 7 each pass was made faster and at a max cost of 30% CPU consumption. Predictably the Nexus 7 is able to gather more data with less resource consumption than the virtual device. However, the cost of this performance increase was at the expense of the variance between the actual CPU usage from the target usage.

The increase in variance can be attributed to the additional cores provided by the Nexus 7. Figure 13 shows the effect of the varying number of active cores on DroidMonitor's CPU consumption. The number of active cores is based on information gathered from proc/stat and the normal processes executing in user mode. As you can see the number of cores currently running, along with the changing number of active cores, increases or decreases the variance of the resource consumption.

Figure 14 shows DroidMonitor's resource consumption on a Nexus 7 while playing Hill Climb Racing [11], a top free Android game. DroidMonitor is unaffected by the game and maintains its target consumption.

D. Network Monitoring

Our network monitor relied on Tcpdump [10] to collect packet traces of the applications running. The CPU usage of Tcpdump was found to be negligible, unless the device is

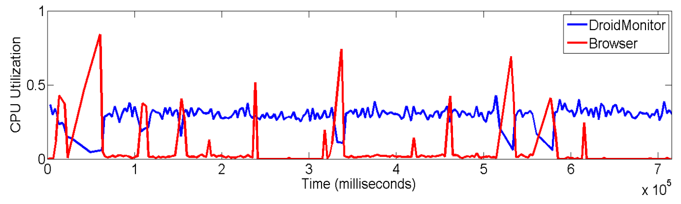

Fig. 8. PI Controller: T = 30, K = 100, Browsing the Internet

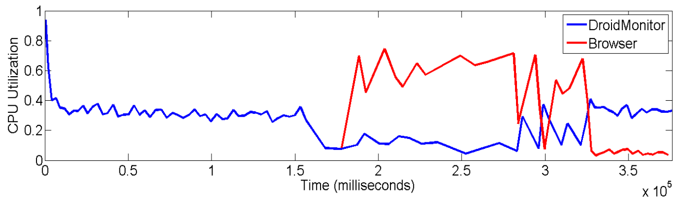

Fig. 9. PI Controller: T = 30, K = Dynamic, Browsing the Internet

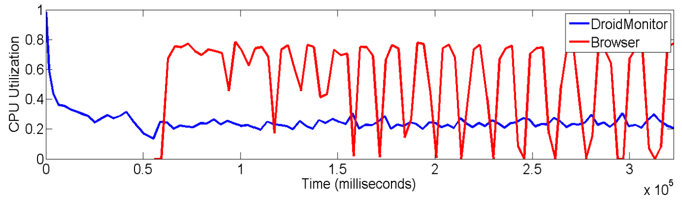

Fig. 10. PI Controller: T = 30, K = Dynamic, CPU Consumer

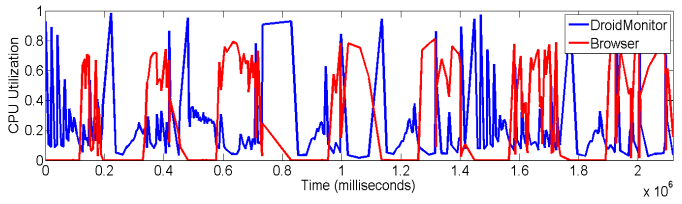

Fig. 11. PID Controller: P = 1, I = 1, D = 10, T = 30

under extreme network load. The highest resource consumption issue occurred while writing the packets gathered to a file. This issue was dealt with in a method similar to other monitors within DroidMonitor, by controlling the ON and OFF periods of the monitor. While Tcpdump was constantly active, the act of writing the gathered packets to a file only took place within the feedback monitor's allotted on period. Tcpdump's output is captured in a buffer and this buffer is emptied by DroidMonitor, if the buffer is full and no data can be written to it, then the information is dropped. This allows for Droidmonitor to catch only a percentage of the captured packets that is consistent with the amount of resources the monitor is allowed to use. Figure 15 shows DroidMonitor's CPU consumption while gathering network packets during a heavy Internet browsing session. DroidMonitor lowers its own CPU usage to accommodate Tcpdump's usage in order to maintain the set target consumption.

E. Testing DroidMonitor with Other Applications

Along with DroidMonitor two other applications were created in order to facilitate a way to test DroidMonitor.

The first of these applications is called CPU Consumer. CPU Consumer was created in order to test how DroidMonitor reacts to other applications consuming large amounts of resources in a controlled manner. The application oscillates between high CPU usage and low CPU usage, at first the OFF period of CPU Consumer starts very quickly and slowly increases to longer and longer OFF intervals. This allows for standardized testing of the controllers response to other applications running asynchronously.

The second application for testing is a very simple game that involves 'popping' falling balls with your finger. Two versions of this game were created, one version is simply the game, and the second version is infected with a Trojan horse. The Trojan horse simply gathers data of the users' current location and sends the information to a remote server. While not directly related to what was being tested with DroidMonitor, the addition to the game was made to see if a difference could be found among the two different versions of an application that could be detected with the two monitors. The infected version of the application was modeled after Android.Tapsnake, a malicious geo-locating application found in the wild [12]. More research needs to be done on analyzing the data gathered, but a small test can show that the concept of monitoring an Android device has validity as a solution to detecting malicious applications on the device.

While the main purpose of DroidMonitor was to create an application to allow users to monitor and collect data on their device, we were interested in DroidMonitors ability to detect malicious applications on the device. After running both clean and infected versions of Poppers on identical emulators while simultaneously gathering information with DroidMonitor it was clear that a difference could be seen between the two different versions. Not only were the hashes different, but the resource consumption was noticeably affected as well. More tests need to be carried out, but it is our hypothesis that peer pressure could be used to detect infected applications on mobile devices in a cloud network.

V. Conclusions

In this paper, we presented a control theoretic adaptive monitoring tool for the Android platform that we termed DroidMonitor. DroidMonitor is able to effectively monitor an Android device while being aware of the limited resources of the device. Monitors can easily be added to DroidMonitor and can be controlled by a number of controllers. We have investigated a number of traditional controllers and investigated their performance in terms of their stability and response behavior under different stimuli. We have found that the dynamic PI controller achieved our best performance results. We have experienced some difficulty tuning the weights of the PID controller. Our results are validated through emulation in virtual devices and implementation on the Nexus 7 tablet.

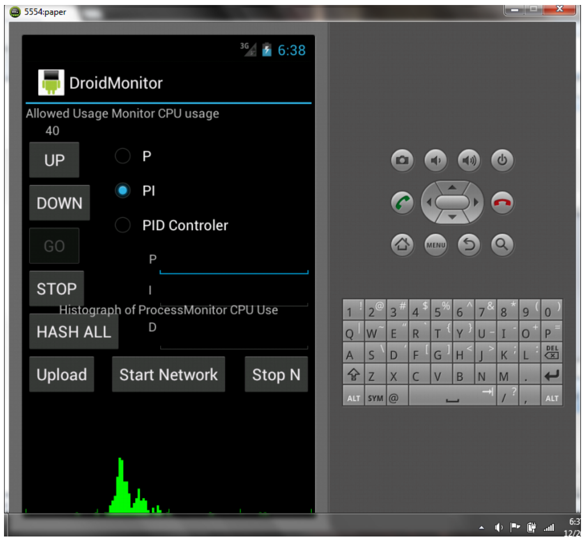

Fig. 12. DroidMonitor Interface

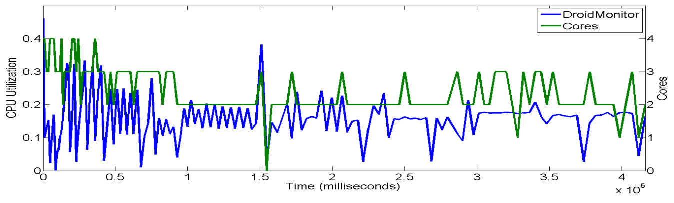

Fig. 13. PI Controller: T = 15, K = Dynamic

Acknowledgments

This research is funded by a grant from the Air Force Research Laboratory and by NSF award #1149397.

References

[1] "Malicious mobile threats report 2010/2011," http://www.juniper.net/us/en/loca/pdf/whitepapers/2000415-en.pdf, 2011.

[2] "Os monitor," available from https://play.google.com/store, August 2011.

[3] "Android assistant," available from https://play.google.com/store, September 2012.

[4] "Advanced task killer," available from https://play.google.com/store, July 2011.

[5] Xuxian Jiang Yajin Zhou, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, San Francisco, California, May 2012.

[6] Wu Zhou, Yajin Zhou, Xuxian Jiang and Peng Ning, "DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces," in *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, San Antonio, Texas, Feburary 2012.

[7] Iker Burguera, Urko Zurutuza and Simin Nadjm-Tehrani, "Crowdroid: Behavior-based Malware Detection System for Android," in *Proceeding SPSM '11 Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile devices*, New York, New York, October 2011.

[8] William Enck, Peter Gilbert, Byung-gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel and Anmol N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, October 2010.

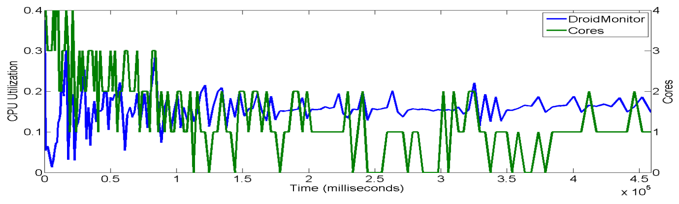

Fig. 14. PI Controller: T = 15, K = Dynamic, Browsing the Internet

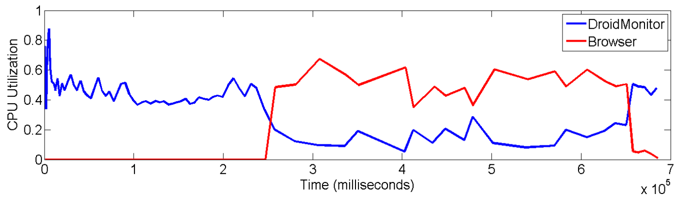

Fig. 15. PI Controller: T = 40, K = Dynamic, Browsing the Internet

[9] William Enck, Machigar Ongtang and Patrick McDaniel, "On Lightweight Mobile Phone Application Certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, Chicago, Illinois, November 2009.

[10] "Tcpdump & libpcap public repository," http://www.tcpdump.org/.

[11] "Hill climb racing," available from https://play.google.com/store, November 2012.

[12] "Android.tapsnake," http://www.symantec.com/security_response/writeup.jsp?docid=2010-081214-2657-99, August 2010.