ENERGY PROFILING & CONTROL FOR ANDROID DEVICES

by

Rahul Murmuria
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

Angelos Stavrou — Dr. Angelos Stavrou, Dissertation Director

Brian L. Mark — Dr. Brian Mark, Committee Member

Duminda Wijesekera — Dr. Duminda Wijesekera, Committee Member

Andre Manitius, Department Chair

Dr. Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering

Date: 01/03/2011

Fall Semester 2010
George Mason University
Fairfax, VA

Energy Profiling & Control for Android Devices

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Rahul Murmuria
Bachelor of Science
National Institute of Technology, Jaipur, India, 2008

Director: Dr. Angelos Stavrou, Professor
Department of Electrical and Computer Engineering

Fall Semester 2010
George Mason University
Fairfax, VA

Acknowledgments

I offer my sincerest gratitude to my advisor, Dr Angelos Stavrou, who has supported me throughout my thesis with his patience and knowledge whilst allowing me the room to work in my own way. I attribute the level of my Masters degree to his encouragement and effort and without him this thesis, too, would not have been completed or written.

I am also very grateful to the guidance and support from Dr. Brian Mark, who is my official graduate advisor and mentor. It is with his advise that I setup my entire masters programme and my association with Dr. Angelos Stavrou was also brought about due to his recommendations.

I am indebted to many of my colleagues to support me. I would like to specially mention Zhaohui Wang who has collaborated with me on various research elements in this funded project. Outside of the project tasks, he has also been a strong support and acted like a true senior as I transition from being a masters student into a PhD student. I also thank Muhammad Abdulla for collaborating with me on some of the tasks and giving me advice on many of the technical issues.

I would also like to mention Anant Narayanan, my friend and collaborator with whom I began my journey into Operating System Design and other computer science subjects. He is one of my close inspirations that led me on to this path of being a researcher. Working on the operating system projects with him provided me with enough mileage to be able to grab the opportunities I have coming towards me today.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project.

Table of Contents

Page

List of Tables vi
List of Figures vii
Abstract viii
1 Introduction 1
2 Background and Related Work 4
2.1 Survey of Mobile Platforms 4
2.1.1 Attacks on Mobile Devices 5
2.2 Energy Accountability in Modern Operating Systems 6
3 Challenges 9
4 System Goals and Architecture 11
4.1 Security Goals 11
4.2 Resource Control 12
4.2.1 Linux Control Groups 12
4.2.2 Implementation of Wakelocks for Power Management in Android Platform 13
4.2.3 Energy Resource Container Design 13
4.3 Virtualization 15
4.4 System Implementation 18
4.4.1 Our Approach 18
4.4.2 Kernel subsystem driver design 19
4.4.3 Energy model 19
4.4.4 Proc filesystem exports 21
4.4.5 Userspace solution using application design 21
5 Experimental Results 22
5.1 Real-time Application Monitoring and Profiling 22
6 Future Work 26
6.1 Energy Policy Enforcement 26
6.2 Efficient User & Device Profiling 27

7 Conclusions 29
A CGroups 30
A.1 Linux CGroups 31
Bibliography 32

List of Tables

Table		Page
2.1	A Comparison of Mobile Platforms.	5
6.1	Sample data illustration for input to dynamic hobbets	28

List of Figures

Figure		Page
4.1	System components.	14
4.2	Overall System Architecture.	16
4.3	Proc Filesystem Exports	21
5.1	Per-process CPU Utilization graph.	23
5.2	Overall battery Utilization graph.	23
5.3	Metering process CPU utilization graph for different times.	24
5.4	Kernel process CPU utilization graph for different times.	25

Abstract

ENERGY PROFILING & CONTROL FOR ANDROID DEVICES

Rahul Murmuria

George Mason University, 2010

Thesis Director: Dr. Angelos Stavrou

Nowadays, smart-phone devices provide increased accessibility and they are equipped with a wealth of standard capabilities including but not limited to touchscreen display, WiFi communications, bluetooth, audio, and GPS. These increased capabilities enable users to perform activities that go beyond mere phone calls: Internet browsing, email, games, pictures, audiobooks are just a few of the growing list of functionality that is currently supported by modern phones. Unfortunately, this increased functionality has high resource consumption requirements that incurs a direct impact on the battery life of all hand-held devices. Google's build-in battery display employs a simple, linear model to calculate the energy requirements of the active devices. Moreover, Android does not provide any mechanisms to control or even meter the resource utilization for individual processes and devices. The existing model appears to be inadequate in providing proper component based analysis for battery consumption such that it is possible to profile and control applications based on their device usage pattern.

In this thesis research, we will attempt to provide a more precise model of measuring and policing the power consumption in Android-equipped hand-held devices. To that end, we plan to design and implement a kernel subsystem to calculate and assign live adaptive weights to each hardware device component within the Linux kernel. Our approach will use moving averages to identify the relative impact of each device usage on the battery consumption curve. This will be done taking into consideration other factors including but not limited to change in temperature and heat over time. In addition, we will further provide mechanisms to estimate the battery consumption of each application and/or each type of task in the Dalvik VM and in the Kernel. Our goal is to enable the user to maximize the availability of critical functions like, for instance, making calls and minimize the use of battery consuming tasks like, for instance, continuous GPS location updates on Google Maps.

Chapter 1: Introduction

Recent advancements in hardware have increased the computing power, memory, storage, and wireless connectivity of handheld mobile devices. Smart phone devices are used for everyday activities that range from Maps and Geo-location tagging to banking. Indeed, these new hand-held devices are capable of carrying significant amount of both personal and professional data including documents thus extending the operations that we can perform from desktop to small-factor devices. Unfortunately, this reliance on hand-held devices have made them an attractive target for applications and new mobile application markets have spawn for the different types of devices. At the same time, these new devices have become the target of malicious attackers that have shifted their attention from desktop systems to malware and malicious software for hand-held mobile devices.

Most mobile phone devices today are equipped with a phone, web browser, music player, camera, and a horde of other applications and services. Google Android, NeoFreeRunner, Nokia Maemo, iPhone OS and Windows Phone OS are some noteworthy hand-held device platforms capable of performing most of the functions previously found only in full-fledged desktop operating systems. Usability of such devices is further enhanced by the availability of third-party applications that can be purchased or freely downloaded by users from online application stores or developer websites. This possibility for greater functionality and convenience, however, it also exposes the user to a greater risk from malicious software programs. While mobile applications that can be downloaded from untrusted third-party sites are commonly regarded as the main source of such malicious software, security risks can also come from vendor-certified app stores, as it is difficult for the vendors to thoroughly test thousands of applications whose behavior could potentially be made time and location dependent. Further, even the built-in applications that come with such wireless mobile devices may contain security holes that need to be addressed.

Hand-held wireless devices are different than conventional computing platforms, such as desktop computers, in several aspects. Firstly, such devices have very limited resources in terms of computing power, batteries, memory, to name a few. A faulty or malicious program targeting these weaknesses, for example, can easily abuse the unregulated access to the battery through excessive use of CPU or radio communication. Secondly, hardware parts are tightly integrated, which makes it difficult to identify the source of an abnormal condition. For instance, if a program is able to increase the GPS intervals of communication, it is not easy for the end-user to identify the cause of power consumption and take action. Thirdly, due to mobile nature of such devices, and lack of built-in protection against malicious software through the underlying operating system or through third-party anti-virus programs, malicious programs can spread faster.

In this thesis, we introduce a kernel-based approach to meter, profile, and police the power consumption of applications that run in Android devices. Our goal is to provide increased fault tolerance security by quantifying the power consumption that is related with a specific process or a set of processes including operations of this process that involve kernel subsystems and devices. We do so by making each process accountable for the use of all resources in the hand-held device including CPU, communications, GPS, USB, storage, display, touchscreen among others. In this thesis, we have competed the accountability and profiling components but we have not fully implemented a policy enforcement module to prevent misuse of battery. Therefore, although we can detect energy changes and attribute them to processes, we cannot prevent those processes from consuming more energy. In this regard, we focus on two aspects of the energy control system at the same time: **resource management** and **process control** through Kernel and user-space modifications.

We focus on resource usage accounting, application profiling, and resource provisioning for processes or process groups. In particular, we concentrate on **battery management**. The reasons are that the battery is among the most important resources for handheld devices and that there is little OS-level support for battery profiling and provisioning, in contrast to other types of resources such as CPU, memory, etc. Our ultimate goal is to

provide efficient and effective security mechanisms that support a wide variety of security policies to create a secure execution environment for mobile devices. We choose Google's Android platform [1] to implement our solutions. This is primarily due to the open source nature of the Android platform and the expected increase in the market share of Android based mobile devices.

Summary of contributions:

- Design & Implement an accurate model for accounting and policing energy consumption. We do so using a novel kernel subsystem to estimate energy and meter energy consumption of each process.
- Meter the per-process CPU & Device utilization over time. To that end, we identify the relative impact of each device component on energy consumption.

Chapter 2: Background and Related Work

In this chapter we first go over popular mobile platforms, such as iPhone, Palm, and Google Android, from the perspective of security enforcement for software applications. Then we go over common attack models on mobile devices. Afterwards, we discuss related literature on battery accounting and control. Finally, we go over existing virtualization solutions for conventional computing platforms, highlighting advantages and disadvantages, and the suitability of different virtualization solutions for mobile platforms.

2.1 Survey of Mobile Platforms

In this survey, we discuss three common mobile platforms: iPhone, Palm, and Android.

The iPhone mobile platform released by Apple uses an application model where software programs are restricted through "*application sandboxes*". Application sandboxes are used to limit an application's access to files, preferences, and other hardware resources. The sandbox helps to reduce the damage caused compromised non-malicious software programs. A malicious application, however, can still stage attacks on the phone.

The Palm platform uses Linux kernel with proprietary extensions. A discussion of the vulnerabilities of the platform is discussed in [2]. Due to the propriety nature of the iPhone and Palm platforms, in this work we focus on the open source Google Android project, which uses a slightly modified Linux kernel as its operating system core.

Google Android platform built on a modified Linux kernel, and provides libraries for C, audio, video, database, and network communications. Android uses Dalvik VM for executing and containing programs written in Java. The VM expects a bytecode different from Sun's official bytecode. The library is based on a subset of Apache's Harmony toolkit.

Application control is achieved through the policy enforcement mechanism of the Dalvik

Table 2.1: A Comparison of Mobile Platforms.

⇓ Feature \ Platform ⇒	**Android**	**iPhone**	**Palm**
Development platform	Android SDK	iPhone SDK	Palm WebOS
Open source?	Yes	No	No
Supported languages	Java	Objective-C	Javascript/HTML
Virtualization	Dalvik VM	No	No
IDE	Eclipse	Xcode	Eclipse
Emulation	Android Dev. Tools	iPhone SDK	Palm SDK
VM monitoring support	Mainly for debugging	N/A	N/A

VM. Applications written in Java that runs inside Dalvik can only use the set of functionalities provided by the Java API. However, the need for application control and the need for providing broad system level services are often contradictory. A malicious program can always ask for more permission that it needs, and users cannot be expect to make correct decisions all the time. Further, the Android platform allows the use of native C/C++ libraries that come with Java applications. This opens more opportunities for malicious applications to attack the system.

Table 2.1 gives a comparison of features of different platforms.

2.1.1 Attacks on Mobile Devices

Generally, attacks against mobile devices can be categorized to two types: *disruptive* and *non-disruptive*. Disruptive attacks are attacks that make the device to a non-usable state. For example, DoS attack against the battery, blanking the screen to prevent the user's operation, or even overclock the CPU causing hardware damages. Non-disruptive attacks mainly focus on stealthily information capturing and transmitting user data to external adversaries. Such attacks do not affect user experiences on the devices but will leak sensitive or confidential information, such as user contacts, user location through GPS, keyboard input, screen captures, audio/video recording, etc.

In terms of attack channels, we can group attacks on mobile devices as follows:

- Vulnerabilities in built-in applications, such as buffer overflow vulnerabilities [3], etc.

- Downloaded malicious software.
- Attacks through input channels, including USB connections, wireless communication, etc.

The motivations for attacks could be due to financial, military, or personal reasons. Attack types could also be different, such as traffic re-routing, spying through microphone and camera, stealing data, modifying system messages, altering audio and graphics output, attacks on resources such as battery drain, packet drops, high CPU usage, or even explicit attacks, such as playing sound or video.

To aid our studies, and to show proof of concept demonstration of possible attacks, we have developed programs that perform most of the types of attacks that we have described above, such as blanking out device display, recording sound and video, sending files to remote locations, etc. We also have initial success in attacking certain hand-held devices through USB connections. The lack of deployed USB defenses or detection mechanisms empowers the attacks to remain stealthy. The only instance of USB-borne threats is flash drive viruses spreading from USB files. However, the new smart-phones are capable of accomplishing a much more powerful and widespread propagation of malware.

2.2 Energy Accountability in Modern Operating Systems

As we discussed earlier in the introduction, one of the main difference between mobile devices and conventional computing platforms is resource availability. The most pronounced among them is battery. Defective applications or defective updates to previously functional applications can cause the hardware devices on the phone to overtax the battery. Deliberate or accidental, these applications hijack the WiFi, Bluetooth or display of the device and eventually cause a denial of service attack [4]. Another possible attack vector for battery exhaustion is to taint the mobile device's ability to sense network signals, thereby causing it to go into signal searching mode too often, and resulting battery exhaustion.

There is existing literature on design proposals for an intrusion detection system based

on per-process battery consumption information [5]. Their results are purely mathematical, and the proposed model is based on artificial neural networks.

There is also a tool for power-profiling, named pTop [6], which uses in-kernel code to collect per-process CPU, Network and Disk usage statistics. Their power consumption calculation is based on an energy-model that uses specification provided by hardware vendors and generates statistics over a sampling time window. This design employs the use of extensive data collection and analysis which is costly on system resources.

In another research initiative, the goal was to dynamically tune a wifi receiver's energy-saving strategy depending on the application behavior and key network parameters [7]. Here, the researchers modified the process scheduling algorithm in accordance with the hardware utilization from each process in order to achieve power saving for the entire system. While their work demonstrates that it is possible to restructure process scheduler's wait queue in accordance with each process's hardware usage, the goal here was overall system energy optimization, and not per-process energy profiling, hardware access rate control and anomaly detection.

It is also possible to determine battery-consumption information during compile time [8, 9] by application designers during their quality assurance and testing phase but these estimates are purely under lab conditions and are measured offline through code analysis, rather than a live runtime calculation that might adapt to heat and hardware lifetime.

The use of linear regression model with static weights for hardware components' relative rate of battery consumption as described in [10], is further questionable. Battery discharge is never ideal, and the process scheduler has to take the non-linearities into account, in order to guarantee good battery usage estimations [11]. It is inaccurate to make a direct correlation between currently running processes and the instantaneous fall in battery levels recorded by the operating system.

In order to defend against these denial of service attacks, in our work we group relevant processes into kernel-space process containers which isolate and then enforce resource limits on each of these groups, depending upon configuration. The kernel has CGroups or Control

Groups [12] which is a module that helps create these process groups. We implement a subsystem of control groups which uses per-process battery consumption information and provide an interface to set limits to the battery consumed by a container. The advantage of basing our battery process groups on this subsystem infrastructure is that there is a consistency in managing the Linux kernel with respect to resource counting.

Chapter 3: Challenges

Why don't we have accurate energy attribution system for Unix processes? To answer this question, we need to look carefully at what is required to build an energy accounting system and identify the potential pitfalls that can lead to miscalculations or lack of attribution.

The main obstacle for accurate metering of energy resources is the current lack of attribution of usage of the device subsystems including but not limited to GPRS, WiFi, bluetooth, display, packet network operations, GPS. Indeed, processes that utilized these devices do not get "charged" for the energy that these subsystems consume to provide their services to processes. For instance, the energy usage of the WiFi subsystem to transmit and receive packets has to be attributed to the corresponding processes that benefit from this communication.

Unfortunately, the problem is not as simple. What happens in the case of shared resources and kernel subsystems? Of course, almost all of the kernel subsystems are shared which means that we have to identify a mechanism to "share" the utilization of each of the device's subsystems to the corresponding processes. This immediately opens up the issue of what type of sharing we will perform. We do not want to penalize processes that have light usage of a subsystem by dividing the energy equally among processes. We need to identify a more fine-grained mechanism to attribute usage of subsystems and their corresponding energy consumption for accurate accounting.

Another issue is that devices do not drain power equally over all time intervals and the actual consumption depends on many different factors including the intensity of the usage for the different device subsystems. As an example, the WiFi driver requires more power in environments with poor signal strength and can quickly drain the battery of the device. This change in "intensity" of the energy consumption for device subsystems has to be accounted for and attributed to the processes that use that device.

Finally, it is clear that there is a need for a more dynamic calculation of power consumption. Therefore, the more frequent we meter the resources the more accurate consumption attribution we can expect. At the same time, the more frequent we meter the more power have to consume for the metering application itself. This creates a relation between accuracy and metering consumption akin to the Heisenberg uncertainty principle in particle physics which states that certain pairs of physical properties, such as position and momentum, cannot be simultaneously known to arbitrarily high precision. In our case too, accuracy cannot be known in arbitrarily high precision. The more the required precision, the more power we have to consume in terms of measurements.

Chapter 4: System Goals and Architecture

In this section, we outline system goals and architecture for mobile device platforms. Since we focus on Google's Android platform in our study, as we mentioned earlier, we first go over the security design of the Android platform.

Under Android's security architecture, processes are isolated in Google's custom JVM, *Dalvik*, and application programs or services written in Java can only get system services through a set of Java API exposed by the JVM. Applications are installed with default permission set, and can request for more permissions at the installation stage. Each application is assigned a user ID under which it will be executed, and resource access is enforced through the underlying Linux kernel based on user and group ID information. Java applications can be augmented through C/C++ libraries using JNI, but the same set of permissions are enforced for resource access.

Different permissions are given for system resources based on their security significance. For example, applications can request access for the camera or the audio recording device and if these permissions are granted at install phase, application can access camera and microphone readings through Java APIs.

4.1 Security Goals

Our eventual goal is to provide flexible security mechanisms to support a wide range of security policies. An example set of security policies may include "No app can access display framebuffer when it is not active", or "Only the 'Phone' app is allowed to run if the battery level is below 5%", etc.

As we discussed earlier, the main source of security problems is the conflict between the user's desire for more functionalities and the resulting exposure to more security risks. It

is infeasible, or at least undesirable, to limit users' ability to extend functionalities through installing more applications.

An example is the aforementioned prohibition of framebuffer access for third-party applications under Android. While there is a clear security risk in allowing applications to get access to the framebuffer, there could be valid applications that need to be able to read the display framebuffer to get a screen capture, or to start a VNC session to be viewed from outside.

As we can see, there is a direct conflict between functionality and security here. To address this, we propose the use of **process control** and **resource management**. By providing a contained execution environment, we address such conflicts by granting application what they need, without compromising system security. For example, a request for the display device can be granted to an application by assigning a virtual display device that the application can manipulate however it wants without being able to access or modify other applications' displays. In this regard, we view our approach as an extension to the security mechanisms currently in place under Android platform.

4.2 Resource Control

Both virtualization/isolation solutions that we have investigated, OpenVZ and Linux Control Group (CGroups), have their own resource control mechanisms for CPU, memory, network, etc. Due to the fact the CGroups are readily available in current Linux kernels, and that there no official patches of OpenVZ for recent Linux kernels, we mainly focus on Linux Control groups in our discussion of resource control mechanisms.

4.2.1 Linux Control Groups

Control Groups [12] provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behavior. These groups have very low overhead as compared to traditional virtualization or jailing techniques. Groups management is available through simplified user-space file operations to control access to

the CPU, memory, network, input/output character devices and filesystem. The processes that appear to be malicious or resource exhausting, can be placed into a frozen or suspended state. These suspended group of processes can be ported to a different computing device and can be re-played in a protected environment for performing analysis.

Since this system was originally implemented in Linux for traditional computing platforms, they do not have support for battery management, an important resource for handheld mobile devices. In our research, we extend CGroups to support battery usage accounting through a battery management subsystem.

4.2.2 Implementation of Wakelocks for Power Management in Android Platform

The Android platform needed a single framework for micromanaging the power suspend for each hardware component. Google Android team has implemented an aggressive power management policy on top of the Linux power management, in which "wakelocks" are used by system processes, in order to keep each hardware component from going back to low-power states, idle and suspend. For instance, it is possible to hold the wakelock for the wifi driver, and hence let the display, gps and bluetooth go into idle mode, while keeping the wifi module awake. We extended this wakelocks system to determine which process has been using which device driver and for how long.

Under Android platform, the `wake_lock_internal` function from the wakelocks driver is called every time this lock is accessed. To account for we record this call in our thread groups statistics (`tgid_stats`) driver that we will describe shortly.

4.2.3 Energy Resource Container Design

In our Energy Resource Container we implemented the following components:

1. Hobbits driver: Calculates a relative weight for impact on battery consumption for each wakelock.

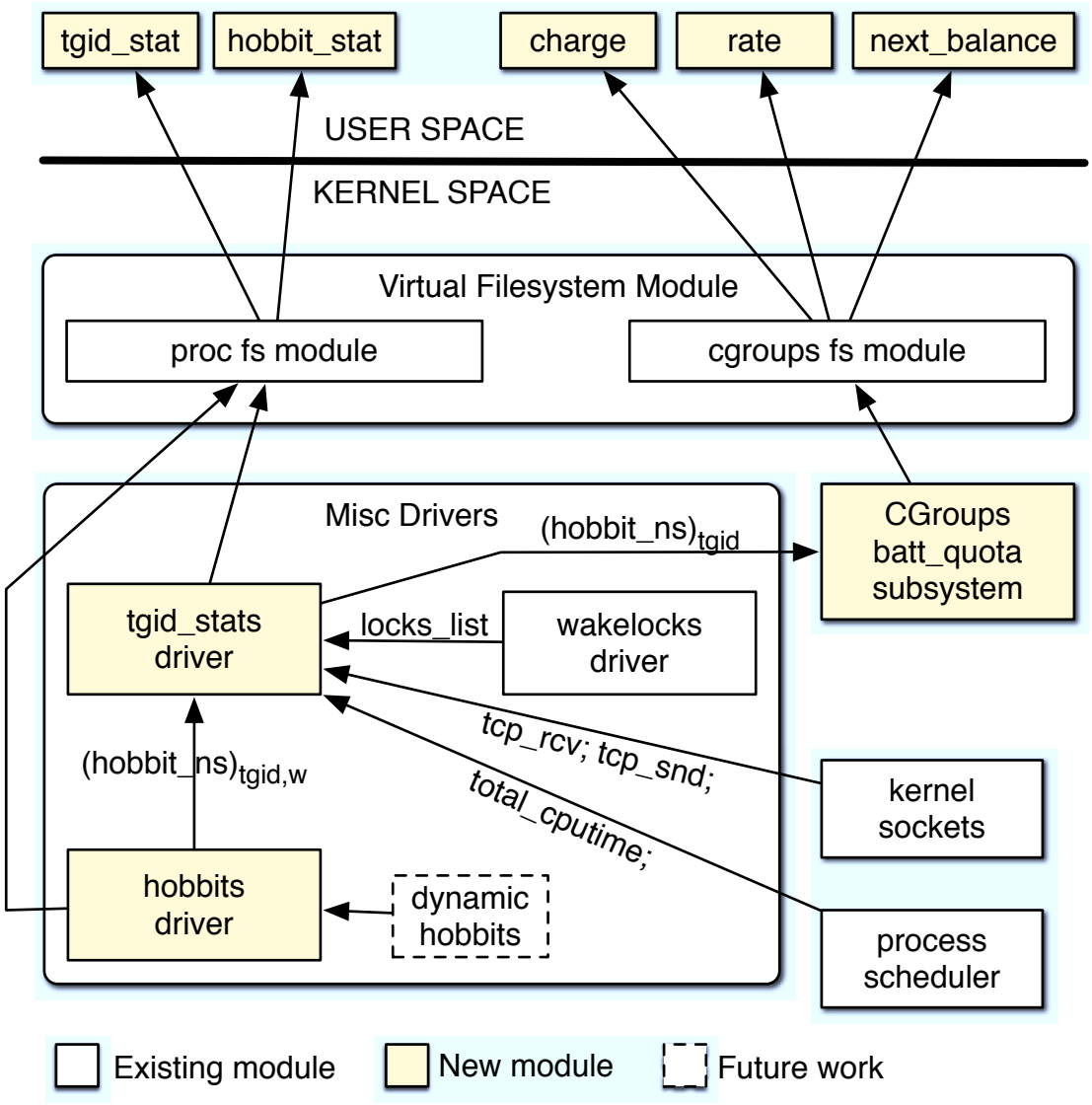

Figure 4.1: System components.

2. `tgid_stats` driver: Maintains lists for each kernel thread group with names of wake-locks and the amount of time the thread group was estimated to have held the lock.

3. (hobbit_ns) calculator: This is a pseudonymous battery consumption value for every thread group that can be used as ratios.

4. proc filesystem exports: All of the hobbit driver and tgid_stats driver generated information is exported to the proc filesystem for users and applications to parse.

5. energy_quota: A CGroups subsystem to control a group of processes based on their hobbit_ns value.

These components are shown in Figure 4.1 and has been described in detail as below.

4.3 Virtualization

To achieve the aforementioned system goals, we considered different virtualization solutions as well. Our design is explained using the following conceptual system architecture as shown in Figure 4.2.

For commonly used application in mobile hand-held devices users expect responsive behavior from the system from application start up to user interaction. In an effort to isolate separate instances of program execution in separate containers, a virtualization solution inevitably introduces execution overhead compared to native program execution. As a result, the main challenge in interactive application support is that the virtualization solution should be light-weight, while providing full virtualization of graphics, audio, and other system resources. Further, it should also support fast and seamless switching of applications running in different containers.

For controlled execution of software applications, we consider the following three different aspects that need to be addressed:

- File system level isolation

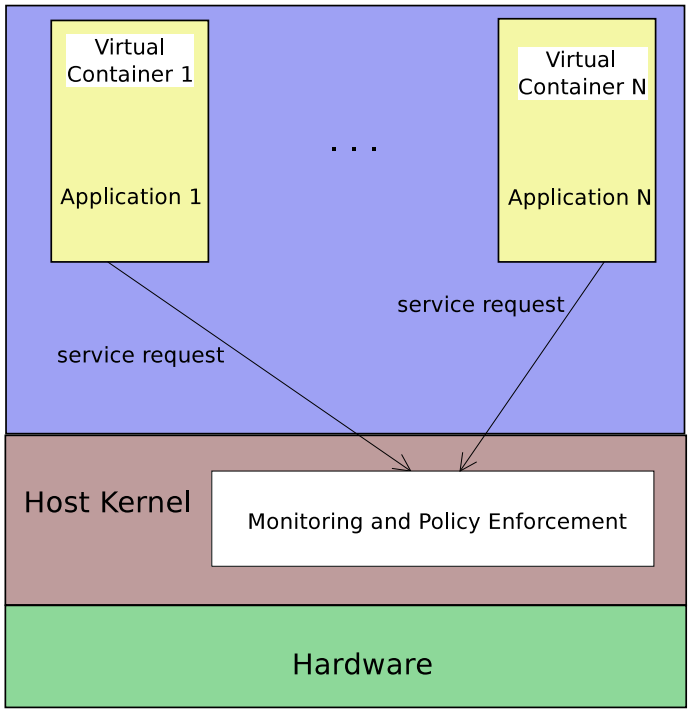

Figure 4.2: Overall System Architecture.

- System resource control
- Virtualized device support

Together, these three mechanisms provide execution isolation and control in terms of file system, processes, and IO operations.

For FS level isolation, we use of `chroot` mechanism to contain application execution under a sub-directory of the host file system. This prevents a malicious application from damaging system files and from unauthorized access to other applications' data, such as preferences, contacts, etc.

Linux *control groups* (cgroups) are used a mechanism to control a group of processes for specialized behavior. One advantage of CGroups compared to OpenVZ is that CGroups exist as part of the Linux kernel, and, unlike OpenVZ, are readily available to use.

Under CGroups mechanism, a cgroup associates a set of tasks with a set of parameters for one or more subsystems. A subsystem is a module that makes use of the task grouping

facilities provided by cgroups to treat groups of tasks in particular ways. A subsystem is typically a "resource controller" that schedules a resource or applies per-cgroup limits, but it may be anything that wants to act on a group of processes, e.g. a virtualization subsystem. Currently available cgroup subsystems including *CPU*, *memory*, *network*, *device access*, etc. For our purpose, we also add a new *battery subsystem* to control process groups based on battery usage.

4.4 System Implementation

4.4.1 Our Approach

In order to provide proper protection against our threat model discussed above, we implemented a model for accounting and policing energy consumption. We followed a two-pronged approach:

- Meter the per-process CPU and Device utilization over time
- Identify the relative impact of each device component on energy consumption

Metering per-process device utilization

Google's Wakelocks are discussed under Background earlier in this paper. These wakelocks are meant to keep the hardware device from going to power save mode. Everytime a process needs access to a device, it asks the OS to get a lock (its a shared lock, not an exclusive lock).

Our module watches these calls and logs which process is calling which lock, and which lock got which processes at a given time. From this log, the 'uptime' is calculated.

We still need 'intensity' of operation for each device. For example, a process could run the CPU at lower frequency. Or, two processes could use the network, and use it at different rates. Therefore, we get CPU and Network intensity information into our driver by pulling this information from the CPU Scheduler and the Kernel Sockets.

Relative impact of devices on energy

Devices effect the energy at varying rates, in comparison to each other. Further, the impact on energy changes for a given device depending upon lots of factors. For example, the cellular signal varies a lot from place to place, and depending upon the user's location, the device could regularly consume more energy than for other users. Same can be said about GPS, where it could be from Satellite or from Network and teh initial seeking time

consumes more power due to its bursty nature, than the regular Location updates to the OS thereafter.

These weights also change from device to device. For example, for Motorola Droid 2, the wifi driver might be more costly on energy than its counterpart in Dell Streak.

To solve this problem, we created a concept of relative energy impact weights, called 'hobbets'.

4.4.2 Kernel subsystem driver design

hobbets Driver

This module maintains a linked list with the name of wakelocks as an index. It uses the following structure:

The rest of the driver comprises of services and setters to manage this structure. Some functions are provided as an API for other kernel modules to access via a header file for this driver. These functions include:

Thread Group Statistics (`tgid_stats`) Driver

This module maintains lists in order to account for the amount of time a thread group holds a lock, and calculate its corresponding `hobbet_ns` score, as a relative value indicating battery consumption. The driver uses the following structures:

4.4.3 Energy model

Each wakelock for a given thread group has a corresponding hobbet value set in the `wakelock_type` structure, that suggests the impact that this lock has on the battery usage every time the lock is held by any process. The `tgid_stat` driver records, in the `wakelock_type` structure, the amount of time a wakelock was active. Next, the `hobbet_ns` value is calculated as:

$$(hobbet\_ns)_{tgid,W} \mathrel{+}= \triangle(uptime)_{tgid,W} * \triangle(intensity)_{tgid,W} * (hobbet)_W$$

$$W = device\,driver\,locks;\ tgid = thread\,group\,(user\,space\,processes)$$

In the above formula, the hobbet value for each wakelock is expected to change periodically. In our current prototype however, this value is absolute and further research is needed to make this attribute adaptive over time to the non-linear battery consumption curve. More details on this are provided in the future work section.

This `hobbet_ns` value is for a given thread group and a given wakelock. To find the corresponding total for a thread group or for a wakelock, we take a sum over all necessary `hobbet_ns` values. This way, we can find the total energy consumed by a given process or a given device driver.

$$(hobbet\_ns)_{tgid} = \sum_{W} (hobbet\_ns)_{tgid,W}$$

$$(hobbet\_ns)_{W} = \sum_{tgid} (hobbet\_ns)_{tgid,W}$$

$$W = wakelock;\ tgid = thread\,group$$

Converting these hobbet_ns values into milliampere-hour (mAh) is possible in the following way:

1. Starting values are recorded for
    - mAh value from the battery driver
    - hobbet_ns value of the thread group or wakelock in question
    - sum of all hobbet_ns values for all the thread groups in the `tgid_stat` list
2. Test is run for any time window as needed. During this time, the kernel will update these structures with new values based on the usage.
3. At the end, the same 3 values are recorded again and the deltas (change in values) are calculated.

Once the above steps are complete, for the given time window, the following simple ratio formula can be used:

$$\triangle B_{tgid} = (\triangle H_{tgid} / \sum_{tgid} \triangle H_{tgid}) * \triangle B$$

$$\triangle B_{W} = (\triangle H_{W} / \sum_{tgid} \triangle H_{tgid}) * \triangle B$$

$$B = battery\,in\,mAh;\ H = hobbet\_ns;\ W = wakelock;\ tgid = thread\,group$$

4.4.4 Proc filesystem exports

All of the hobbet driver and tgid_stats driver generated information is exported to the proc filesystem for users and applications to parse, as illustrated in figure 4.3.

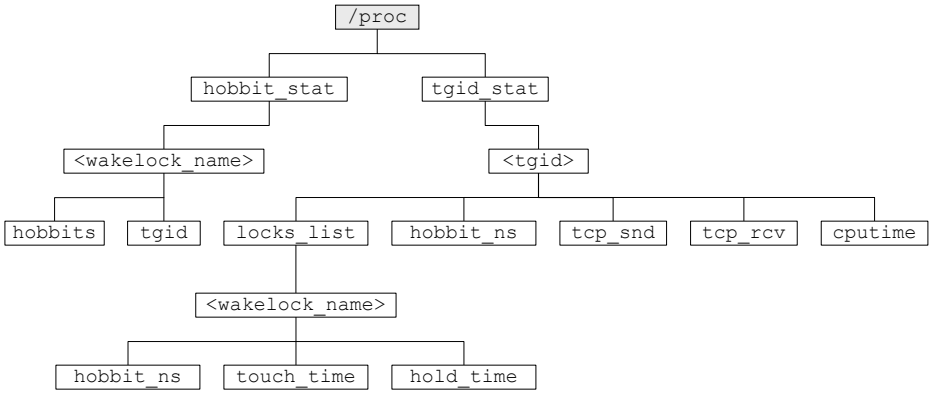

Figure 4.3: Proc Filesystem Exports

4.4.5 Userspace solution using application design

We used SysWatcher application. Design or our logs explained bellow.

Chapter 5: Experimental Results

We were able to produce accurate real-time estimations. In addition, we stored and exported this information to log files and were able to provide real-time analysis over the data generated using an Android log analysis application.

- Per-process CPU utilization
- Per-process Network utilization
- Overall battery usage
- Localize that information (GPS coordinates list)

5.1 Real-time Application Monitoring and Profiling

Our application generates real-time graphs for per-process CPU utilization and overall battery exhaustion as shown in Figure 5.1 and Figure 5.2.

In Figure 5.2 we depict a charging cycle. Notice that by comparing Figure 5.3 and Figure 5.1, we can see that these two different processes were profiled in the same time frame. Each utilized a different amount of CPU ticks. "SysWatcher" is our logger application, and therefore, the CPU ticks in Figure 5.3 illustrates the CPU consumed by the logging application itself.

One other derivation that can be made from the CPU logs is the system power save time for each process. Note in Figure 5.4 that the curve is monotonically decreasing. This indicate that the process is in power-save because `system_server` process is attached to the usage of the device and it only goes to sleep when the device is in power-save mode. However, in most other cases, the only way we can identify if a process is powered down or if

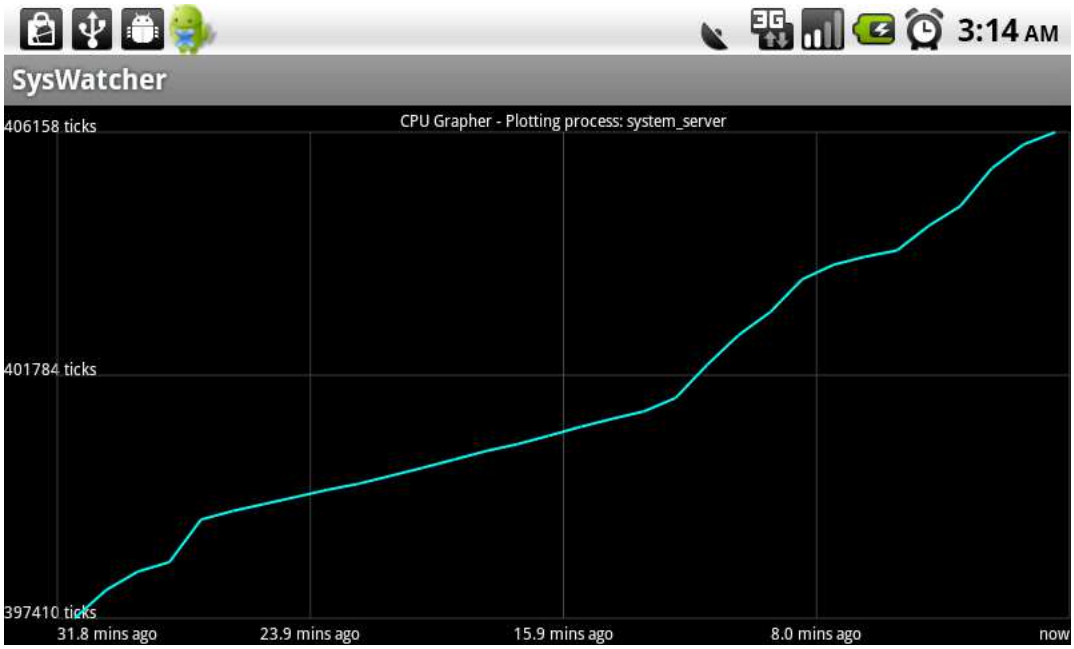

Figure 5.1: Per-process CPU Utilization graph.

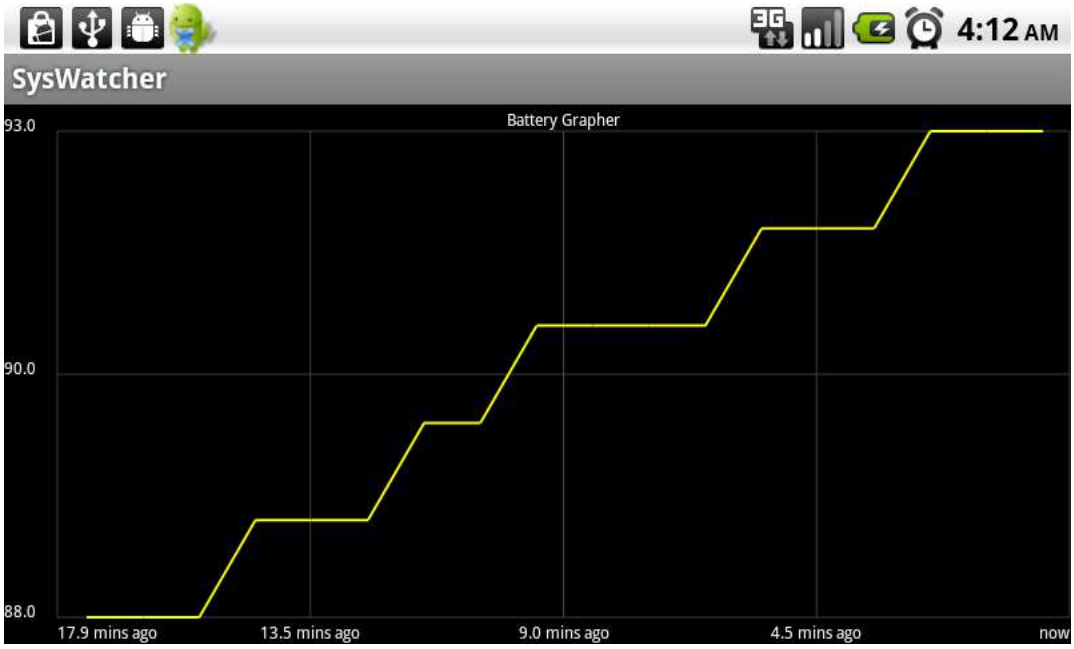

Figure 5.2: Overall battery Utilization graph.

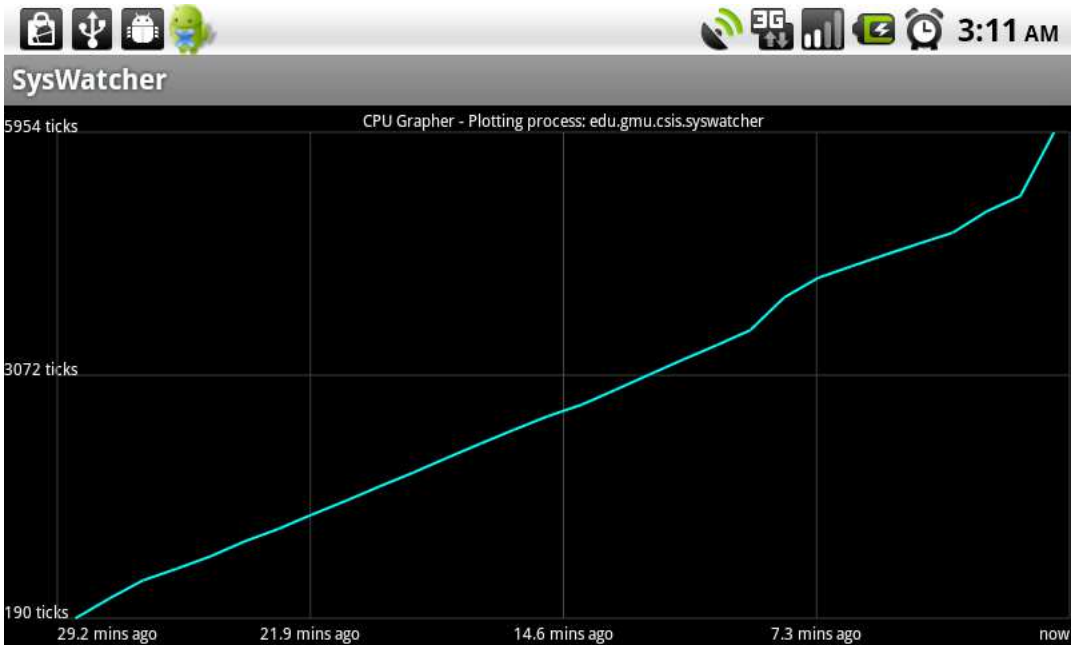

Figure 5.3: Metering process CPU utilization graph.

the entire system went into power save mode, is by looking at the battery exhaustion curve in parallel. Unfortunately, if you are making this analysis in user-space, you cannot detect if you are in power charging mode during which time the battery curve won't be available, and we will not be able to make a clear differentiation between the two phenomena.

Another interesting set of observations that can be made are based on rate of change of battery along with which processes became active and which ones died. We did not implement this view on our SysWatcher application yet, however the same can be generated using the log files as a post-analysis, by exporting the log files to a standard server / laptop.

Sample log files entries:

CPULog:

$2010.December.106 : 48 : 00AMEST, pid90, system_server, ticks417924,$

$cpu0.16999999999999998\%, mem58440KB$

BatteryLog:

$2010.December.106 : 48 : 58AMEST, bat100, volt4.6, temp273, Full$

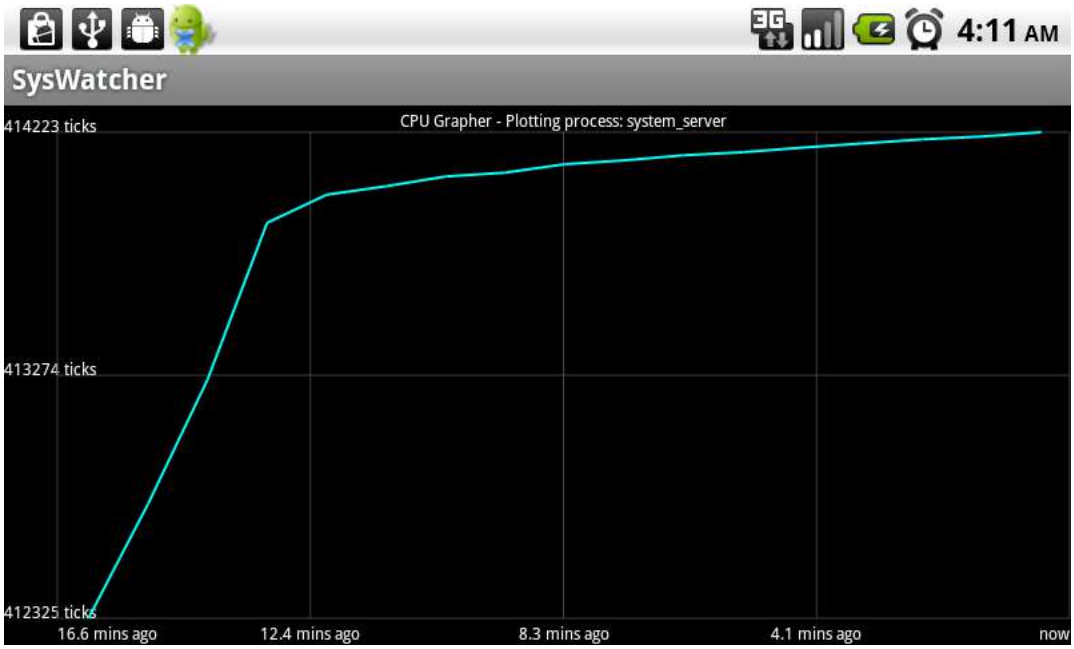

Figure 5.4: Kernel process CPU utilization graph.

GPSLog:

$2010.December.0811 : 19 : 31PMEST, LAT38.84008169174194, LON-77.31281340122223,$

$ALT99.0, ACY8.0$

Chapter 6: Future Work

In this chapter, we describe our plans for future work.

6.1 Energy Policy Enforcement

This is more of an immediate goal. Use CGroups containers to create process groups. Create a new energy control subsystem in cgroups, in order to enforce energy quotas on the process groups.

The cgroup framework needs to be initialized from user-space after every reboot. File operation commands like `mkdir` and `echo` are used to create cgroups and add processes to it. Also, `mount` is used to attach subsystems to a cgroup.

Here, two subsystems, `cpuset` and `batt_quota` have been added to the cgroup `taskset1`. Since they have both been assigned the same hierarchy, the properties that will be defined for both subsystems will apply to all process IDs in the cgroup/taskset1/tasks file.

There are tools and libraries in user-space that help manage creating these cgroups based on preset rules. Also, if the current set of groups need to be maintained across reboots, care has to be taken to save this information in user-space.

On a Linux desktop with standard POSIX libraries, there is `libcgroup` API which can be accessed pragmatically. Further, the `cgred` daemon can be used to allocate child processes to control groups based on some rules set in configuration files. These user-space tools need to be adapted to be usable on a Google Android platform. These user-space functions will need to be implemented as a Dalvik service, or as a Android OS's system component. However, they are not required, in order to test the working of our resource containers and hence this task has been left out of the scope of this study.

Therefore, we aim to create a policy system to control and manage energy provisioning. System administration with energy consumption parameters. To be able to define rules for different classes of applications and functions, the smart-phones can be policed.

6.2 Efficient User & Device Profiling

Overall, our approach can be extended as follows:

- Dynamic hobbet weights: There is a plethora of smart phone devices and more are planned to be released in the near future. The generated power profiles for each application and user have to be adjusted to account for the changes in battery capacity and consumption across devices. To that end, we plan to collect sample energy consumption data and extrapolate the usage for new device using statistics on the variation of hobbet values across different smart-phone devices.

- Associate application energy patterns with users and devices: We plan on associating application energy patterns to specific users and devices by collecting application usage data from a range of different users and devices. This profiling is crucial if we want to be able robustly profile a user. Having a energy profile which is user-centric can be employed for anomaly detection. For instance, we can ferret-out malicious activities and suspicious program behavior, or maybe even predict phone theft due to change in usage patterns.

- Dynamic hobbets for the hobbet driver of the battery subsystem: As we discussed in this thesis, to further increase accuracy, the hobbet value for each wakelock needs to be modified over time. This can be done by averaging the battery consumption curve, with respect to the number of wakelocks active in any time period when the Android mobile device is not in charging mode. The sample data illustration in table 6.1 should explain the concept in more detail.

In table 6.1, we can observe that when `gps` was active, the battery charge fell more sharply. If we started with `wifi`, `display` and `gps` wakelocks having the same hobbet value

Table 6.1: Sample data illustration for input to dynamic hobbets

timestamp (sec)	active wakelocks	charge level (mAh)
1500	[]	1300
2000	[wifi, display]	1250
2500	[wifi, display, gps]	1175

at 2000 seconds, we should have an algorithm to increase the hobbet value of `gps` at 2500 seconds. It must also be considered that the fall in battery charge may not be consistent over multiple test cases and may vary with other factors like temperature.

Chapter 7: Conclusions

In this thesis, we propose a system for accounting of energy consumption for individual Unix process for Android mobile devices. To achieve that, we used a kernel-level subsystem that is two pronged: on one hand it measures the CPU and device usage for each process. On the other hand, it estimates the energy consumption that each of the device subsystems including CPU, display, communications and network sockets, touchscreen, among others. We use the Android wakelocks and device-specific usage to attribute to each process their share of consumed energy from the utilization of the device subsystems.

Our approach departs from the current state-of-the-art because we perform real-time calculations of energy consumption. Previously, engineers relied on the power consumption rating provided by the hardware manufacturers and nearly no real-time calculations were done on actual energy consumption of a device driver. We have implemented our system for Android Linux kernels and we have evaluated its performance using a user-land Android-based application.

Appendix A: CGroups based subsystem for battery quota

Control Groups is used to denote and delineate a set of tasks in the Linux kernel that has been grouped together to better manage their interaction with system hardware. This grouping can be performed based on policies that are decided by either the user, an administrator, or the system based on the parent-child relations. On Android we can set quotas for individual applications by grouping together tasks based on their user id. The policies that are to be used for grouping has been left out of the scope of this work. Our goal here is to demonstrate control groups as a solution in our security model for the Android platform.

The way to interact with cgroups to create, destroy and assign tasks is by using the file system interface. The control groups system allows for process grouping. The underlying mechanism consists of three elements: the control groups, subsystems, and hierarchies. Our *battery* quota subsystem can be enforced upon this process set in the cgroup, and has an interface that allows for assigning of `hobbet_ns` usage limits over the process set in the cgroup associated. The file system has the following elements:

`battery.quota.charge`. This is used to set the absolute battery charge limit to battery consumption. If all the thread groups in a control group container together end up using the given amount of `hobbet_ns`, the tasks are never brought back to the ready queue until the battery is placed into charging mode, or until the cgroup is modified or deleted.

`battery.quota.rate`. This is used to set the percentage rate limit to battery consumption. This interface can also be used to slow down a particular set of tasks. The tasks in the cgroup is not allowed to exhaust the battery unless there are others in the system having an equal piece of it. This is achieved by limiting the `hobbet_ns` of a container relative to the total `hobbet_ns` score of the system.

`battery.next_balance`. This interface determines the overall system `hobbet_ns` value after which the current task is ready to be scheduled again.

A.1 Linux CGroups

Linux *control groups* (cgroups) are used a mechanism to control a group of processes for specialized behavior. One advantage of CGroups compared to OpenVZ is that CGroups exist as part of the Linux kernel, and, unlike OpenVZ, are readily available to use.

Under CGroups mechanism, a cgroup associates a set of tasks with a set of parameters for one or more subsystems. A subsystem is a module that makes use of the task grouping facilities provided by cgroups to treat groups of tasks in particular ways. A subsystem is typically a "resource controller" that schedules a resource or applies per-cgroup limits, but it may be anything that wants to act on a group of processes, e.g. a virtualization subsystem. Currently available cgroup subsystems including *CPU*, *memory*, *network*, *device access*, etc. For our purpose, we also add a new *battery subsystem* to control process groups based on battery usage.

The resource control mechanisms provided by OpenVZ or CGroups can be augmented by Linux operating system's mainstream built-in tools, such as `iptables` or `tc` (traffic control) commands.

Bibliography

Bibliography

[1] Google, "Android open source platform: http://www.android.com." [Online]. Available: http://www.android.com

[2] T. Goovaerts, B. D. Win, B. D. Decker, and W. Joosen, *Assessment of Palm OS Susceptibility to Malicious Code Threats*, 2005.

[3] "Hacking windows ce." [Online]. Available: http://www.phrack.org/issues.html\?issue=63\&id=6\#articlex

[4] B. R. Moyers, J. P. Dunning, R. C. Marchany, and J. G. Tront, "Effects of wi-fi and bluetooth battery exhaustion attacks on mobile devices," *Hawaii International Conference on System Sciences*, vol. 0, pp. 1–9, 2010.

[5] N. Limin, T. Xiaobin, and Y. Baoqun, "Estimation of system power consumption on mobile computing devices," *Computational Intelligence and Security, International Conference on*, vol. 0, pp. 1058–1061, 2007.

[6] T. Do, S. Rawshdeh, and W. Shi, "pTop: A Process-level Power Profiling Tool," in *Proceedings of the 2nd Workshop on Power Aware Computing and Systems (HotPower'09), Big Sky, MT*, 2009.

[7] G. Anastasi, M. Conti, E. Gregori, and A. Passarella, "802.11 power-saving mode for mobile computing in wi-fi hotspots: limitations, enhancements and open issues," *Wirel. Netw.*, vol. 14, no. 6, pp. 745–768, 2008.

[8] N. Azeemi, "Compiler directed battery-aware implementation of mobile applications," *Emerging Technologies, 2006. ICET '06. International Conference on*, pp. 251 –256, nov. 2006.

[9] C. Seo, S. Malek, and N. Medvidovic, "An energy consumption framework for distributed java-based systems," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.* New York, NY, USA: ACM, 2007, pp. 421–424.

[10] D. C. Nash, T. L. Martin, D. S. Ha, and M. S. Hsiao, "Towards an intrusion detection system for battery exhaustion attacks on mobile computing devices," *Pervasive Computing and Communications Workshops, IEEE International Conference on*, vol. 0, pp. 141–145, 2005.

[11] V. Rao, G. Singhal, A. Kumar, and N. Navet, "Battery model for embedded systems," in *VLSID '05: Proceedings of the 18th International Conference on VLSI Design held*

jointly with 4th International Conference on Embedded Systems Design. Washington, DC, USA: IEEE Computer Society, 2005, pp. 105–110.

[12] "Linux kernel cgroups." [Online]. Available: http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

Curriculum Vitae

Rahul Murmuria was born on August 17, 1986, in Kolkata, India. He completed his Bachelor of Technology from National Institute of Technology, Jaipur, India in 2008. He has enrolled into the PhD program at George Mason University and will peruse his doctorate under Dr. Angelos Stavrou after completing his master's degree from the same University. He is employed as a student researcher at the Center for Secure Information Systems at George Mason University since February 2010.