AFRL-RY-WP-TR-2011-1084

DARPA MOBIVISOR: AN ARCHITECTURE FOR HIGH ASSURANCE FOR UNTRUSTED APPLICATIONS ON WIRELESS HANDHELD DEVICES VIA LIGHTWEIGHT VIRTUALIZATION

Anup K. Ghosh and Angelos Stavrou

George Mason University (GMU)

NOVEMBER 2010
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Defense Advanced Research Projects Agency Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2011-1084 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

*//Signature//

JAHN A. LUKE, Project Engineer
Distributed Collaborative Sensor System Technology Branch
Integrated Electronic and Net-Centric Warfare Division

//Signature//

JUAN M. CARBONELL, Chief
Distributed Collaborative Sensor System Technology Branch
Integrated Electronic and Net-Centric Warfare Division

//Signature//

TODD A. KASTLE, Chief
Integrated Electronic and Net-Centric Warfare Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
November 2010	Final	29 September 2009 – 03 October 2010

4. TITLE AND SUBTITLE
DARPA MOBIVISOR: AN ARCHITECTURE FOR HIGH ASSURANCE FOR UNTRUSTED APPLICATIONS ON WIRELESS HANDHELD DEVICES VIA LIGHTWEIGHT VIRTUALIZATION

5a. CONTRACT NUMBER
FA8650-09-C-7956

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62304E

6. AUTHOR(S)
Anup K. Ghosh and Angelos Stavrou

5d. PROJECT NUMBER
ARPR

5e. TASK NUMBER
YT

5f. WORK UNIT NUMBER
ARPRYT0C

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
George Mason University (GMU)
4400 University Drive
Fairfax, VA 22030-4422

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

Defense Advanced Research Projects Agency (DARPA)
3701 N. Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYWC

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2011-1084

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
Case Number: DARPA DISTAR Case 17512; Clearance Date: 27 July 2011. This report contains color.

14. ABSTRACT
This report summarizes the research and activities of the project entitled "An Architecture for High Assurance for Untrusted Applications on Wireless Handheld Devices via Lightweight Virtualization" or simply DARPA MobiVisor. In this work, GMU introduces a containment based security enforcement mechanism designed to contain applications inside virtual containers, separating the running instance of a program from the rest of the system while providing a complete execution environment that supports monitoring, profiling, and controlling applications. A two-fold approach is taken towards these goals: isolation through virtualization and resource management. Isolation addresses the containment of processes at process control and file system levels, whereas resource management handles accounting, profiling, and provisioning of system resources (including CPU, memory, network, battery, and storage, etc). With these mechanisms in place, it is believed that a wide range of security policies can be effectively enforced to provide a secure and lightweight execution environment for applications for "smart" handheld devices. *See reverse for alternate abstract* →

15. SUBJECT TERMS
smart phones, Android, security, virtualization, malware

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Jahn A. Luke
Unclassified	Unclassified	Unclassified	SAR	30	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

14. ABSTRACT (alternate)

With the increased use of “smart” wireless handheld devices, software applications geared toward such mobile platforms are becoming more common. Such devices are being equipped with more than just the ability to place phone calls, and come with applications that provide users with increased functionalities and convenience. However, using third-party application that can be downloaded from app stores or developer websites can also increase the risk of exposure to malicious software programs. Under such circumstances where these devices allow software from third parties to be installed, the onus of protecting the user’s privacy falls directly into the hands of the user himself. There is sufficient incentive for attackers to target these handheld devices, either for gathering sensitive information about the user, or for creating a simple denial of service attack.

In this work we introduce a containment based security enforcement mechanism designed to contain applications inside virtual containers, separating the running instance of a program from the rest of the system while providing a complete execution environment that supports monitoring, profiling, and controlling applications. We take a two-fold approach towards these goals: isolation through virtualization and resource management. Isolation addresses the containment of processes at process control and file system levels, whereas resource management handles accounting, profiling, and provisioning of system resources, including central processing unit (CPU), memory, network, battery, and storage, etc. With these mechanisms in place, we believe a wide range of security policies can be effectively enforced to provide a secure and lightweight execution environment for applications for “smart” handheld devices.

Table of Contents

List of Figures

List of Tables

1. Introduction

With the increase of computing power, memory, storage, and wireless connectivity mobile wireless devices are increasingly used by a wide spectrum of users, in everyday life as well as in military scenarios. These new hand-held devices are capable of carrying significant amount of personal data, like financial bank-related information and important documents. Expectantly, therefore, there has been a significant rise in the threat level around malware and malicious software for hand-held mobile embedded devices. As a result, such mobile devices should be considered as an important part of overall security infrastructure.

Most hand-held mobile 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 increased 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, 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, power supply (batteries), memory, etc. A malicious program targeting these weaknesses, for example, can easily drain 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 example, if a program is able to totally blank out the display, it is not convenient to replace the graphics screen with another to test if it is the problem is with the display screen, or in the active application, or the in the operating system itself. Thirdly, such hand-held devices are often viewed as “small” peripheral clients to the larger desktop or laptop computers, directly connected through a Universal Serial Bus (USB) or other communication channels. This makes it easier for the malicious program in the device to attack and compromise the host. On the other hand, this also makes it easier for the host computer to attack the devices, when, say, a user wants to charge the phone through a USB connection to another person’s computer. Fourthly, 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 project, we introduce light-weight containment based security enforcement mechanisms for mobile wireless devices. Our goal is to provide high assurance of security from malicious third party apps by isolating program execution from the rest of the system inside containers. The framework we developed enables the mobile phone operating system to monitor, profile, control, and enforce policies on applications. In this regard, we focus on two aspects at the same time: process isolation through device virtualization and resource management. Process isolation aims to confine program execution in well-defined virtual containers to block any unauthorized access

and modification of files and other system resources. For resource management, we focus on resource usage accounting, application profiling, and resource provisioning for processes or process groups. 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 [6] to implement our solutions. This is primarily due to the open source nature of the Android platform and expected increase in the market share of Android based mobile devices. Based on the observations above, the purpose of this project is to provide a virtualization based malware protection mechanism for smart phone mobile devices. Our overall aim is to:

1. Provide a lightweight solution suitable for limited resource availability and real-time user interaction.
2. Provide resource isolation, monitoring, profiling, and control for applications.
3. Provide secure bi-directional communication channel between the host computer and the mobile device.

2. Methods, Assumptions, Procedures, and Results

From the perspective of security enforcement for software applications in mobile devices, our approach focuses both on data and resource isolation and management for leap-ahead security technology in hand-held devices. Below is the list of project tasks, work performed, and obtained results.

2.1 Task 1: Survey of Mobile Platforms

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

The iPhone mobile platform released by Apple uses an application model where software programs are restricted through "*application sandboxes*" sometimes called the "walled garden" approach. 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. Details of the implementation, however, are not published.

The Palm platform uses Linux kernel with proprietary extensions. A discussion of the vulnerabilities of the platform is discussed in [7]. 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 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 expected 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 1 gives a comparison of features of different platforms.

Table 1: Comparison of Common 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

2.2 Task 2: Evaluation of Virtualization Techniques

Virtualization was first implemented as a product in 1972 by IBM with System/370 offerings for its mainframe computers. With the increased computing power of desktop systems, virtualization has seen a strong "revival" in recent years. A partial list of virtualization solutions include VMWare products [17], Xen [3], User Mode Linux (UML) [5], VirtualBox [15], and KVM [4] [1].

Virtualization solutions have been used for operating system development and testing, resource consolidation, resource isolation, application debugging, profiling, and monitoring, etc. In the context of security research, the ability to monitor program execution from outside of the program execution environment provides a useful mechanism for detecting malicious program behavior [9, 8]. At the same time, virtualization is emerging as a potential solution to certain classes of security problems particularly around sandboxing malicious code.

With the increasing adoption of "smart phones," mobile wireless devices are becoming part of overall information infrastructure. Most virtualization techniques we have seen so far, however, mainly focus on desktop solutions and there has been little work in mobile device virtualization. Only recently have there been some signs of development efforts towards this direction.

Different virtualization/isolation solutions exist for conventional computing platforms, such as *paravirtualization* (PV), *virtual machines*, *OS-level virtual containers*, and *process control groups*. Virtual machine monitors, like Xen [3], provides a thin, privileged layer on top of hardware to support one or more guest operating systems. Virtual machines, on the other hand, provide an emulated hardware environment from within the operating system. OS-level virtualization can be viewed as "slicing" the host operating system into multiple smaller partitions. Process control groups is not normally considered as a virtualization approach, however, in this context it can provide isolation and resource control for processes.

Regardless of the virtualization approach, from the perspective of mobile devices, virtualization overhead and battery consumption is of paramount interest due to resource limitations and user expectations when starting an application. Table 2 gives a comparison of aforementioned virtualization solutions.

Table 2: Comparison of Virtualization Methods

	Para-Virtualization	**Virtual Machines**	**OS-Level Virtualization**	**Process Control Groups**
Examples	Xen	VMWare, VirtualBox	BSD Jails, OpenVZ	Linux Cgroups
Overhead	Execution: low Startup: high Resource: high	Execution: fair Startup: high Resource: high	Execution: low Startup: fair Resource: low	Execution: low Startup: fair Resource: low
Isolation	Very Good	Good	Good	Implementation Dependent

We can see from the comparison that para-virtualization and virtual machine solutions are not really suitable for mobile hand-held devices due to high resource usage and higher application

[1]Apart from *system virtual machines* mentioned above, the term "virtual machine" has also been used to refer to *process virtual machines*, such as Java Virtual Machine, etc. Unless otherwise noted, by "virtualization" we refer to system virtual machines.

start times. Based on these comparisons, we initially chose to focus on OS-level virtualization and process control groups as potential solutions to be considered for our purposes.

In terms of mobile platforms, there has been little support for virtualization: Palm and iPhone do not have any virtualization at all, while Google's Android platform uses a process virtualization solution using a custom Java Virtual Machine (so-named Dalvik Virtual Machine). Although providing process level virtualization is better than giving full control of the device to applications from a security perspective, the need of the system to restrict process access to system resources and the desire of the process to access system resources as it wishes is often contradictory. *In our approach, we allow application to run in separate virtual containers, while performing resource management and policy enforcement at the host, one level below the virtual execution environment of the process.*

We settled on the Google's Android platform [6] primarily due to its open source software stack. We ported OpenVZ [16], a lightweight virtualization method, for our virtualization solution. In parallel, we also consider Linux CGroups mechanism as a way for process control and resource management solution. While most virtualization solutions focus on platforms based on Intel architecture, most small devices use more power-efficient alternatives such as ARM processors. We completed the port of OpenVZ to the ARM-based Google Android platform to create the first Android kernel that supports OpenVZ virtualization solution to our knowledge. We currently have a functioning OpenVZ Linux kernel that supports the Android platform. We also extended the Linux kernel to include support for battery management in a similar way that has been done in the Linux kernel for CPU, memory, and network.

2.3 Task 3: Identification of Threats on Smart Phones

Broadly speaking, channels of attacks on mobile devices can be categorized as follows:

- Vulnerabilities in built-in applications.
- Malicious downloaded 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 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, and sending files to remote location. We have initial success in attacking certain hand-held devices through USB connections.

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. For these tasks, we acquired Android-based devices and performed initial setup of the development environment. We also have initial success in attacking certain hand-held devices

through USB connections. In addition, we analyzed the attack vectors pertaining to resource exhaustion. We completed the following sub-tasks:

1. Design and develop a set of proof of concept malicious programs: screen blank out, sound beeper, battery drainer, network traffic hog, expose position through GPS tracking. We developed applications that can disrupt the normal operations of a normal phone and can be delivered over the network with the consent of the user.

2. Design and develop malicious applications that can evade the current updating mechanism and user consent to either leak information or disrupt the device. These types of attacks rely on remote exploitation and updating mechanisms and they are more severe threats to the smart phone devices.

3. Profiling applications based on their resource consumption. This will allow us to be able to detect malicious applications or modification of legitimate applications based on behavioral model profiling based on lab and user-collected experience.

2.4 Task 4: Isolation Architecture for Secure Code Execution

The goal of this task is to architect a solution for running untrusted code in containers to isolate the untrusted code from the rest of the platform, while providing a seamless user experience. In this task, we employed a lightweight virtualization mechanism, OpenVZ, for the Google Android platform. The architecture employs layered file systems, providing detection of malicious or anomalous events, and remediation after attack. In addition, we investigated metering resources of the device given to untrusted code. We completed the following sub-tasks:

1. We successfully ported OpenVZ to the ARM platforms that run Google code. The task involved heavy kernel re-programming since both OpenVZ and Google have modified the vanilla Linux Kernel according to their needs. The completion of this sub-task allows us to isolate and meter resources consumed by Android and C/C++ applications. The main challenges in this process were due to independent changes made to the Linux kernel by the Android and OpenVZ projects, and lack of an official port of OpenVZ to ARM-based architectures, on which Android is built. We currently have a running Android platform with kernel-level virtualization based on OpenVZ. We have also been able to port OpenVZ tools for ARM architecture to Android, and prepared OpenVZ templates for the Android platform.

2. Control and Isolate the sharing of critical resources including:

   a. Display

   b. Battery

   c. Communications (Network, Wi-Fi, Bluetooth)

   d. File System Isolation

The Android kernel architecture and the devices used for display are crucial for the design of a control and isolation architecture for mobile devices. Indeed, Android relies on the standard framebuffer device (/dev/fb0 or /dev/graphics/fb0) and driver as described in the linux/fb.h kernel header file, which provides an abstraction for the graphics hardware. The framebuffer device acts as an abstraction of the real the framebuffer of any video hardware and allows application software to access the graphics hardware through a well-defined

interface. This allows software to operate without any knowledge of the underlying hardware beyond well-defined capabilities. The framebuffer driver is usually implemented as a character device under the Linux kernel. To implement display virtualization, we wrote a kernel module that implements the framebuffer operations.

To create multiple framebuffer devices, we insert multiple modules that only differ in driver name (so that more than one driver can be inserted). Assuming two framebuffer devices, /dev/fb1 and /dev/fb2 are created, these two files can be mapped to the /dev/fb0 file of two different execution environments. Applications running inside such environments will see and use such frame buffer devices as normal graphics devices, but they cannot use or modify the framebuffers of other virtual devices or the physical display without an explicit mapping by a privileged user at the host level. Once allocating the video memory and setting the pixel format, the system can use mmap() to map the memory into the process's address space. All writes to the framebuffer are done through this mmapped memory.

As discussed earlier, one of the main differences between mobile devices and conventional computing platforms is their reliance on battery resource availability. 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 can take over communications, including Cell (GSM/EDGE), WiFi, Bluetooth, or display of the device and eventually cause a denial of service through battery starvation. 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. To achieve the isolation we used existing kernel mechanisms modified appropriately for the Android platform: our approach leverages Linux Control Groups to 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. Linux Groups management is available through simplified user-space file operations to control access to the CPU, memory, network, input/output character devices, and file system. All running processes that appear to consume resources beyond their profiled and permitted allotment can be placed into a suspended or resource-controlled state. This prevents them from exhausting the mobile device resources and alerts the user of the presence of a non-conforming application. Processes that are added to the suspended group of processes can potentially be ported to a different computing device and can be re-played in a protected environment for performing analysis.

In addition to the battery and display, we use the existing Google-implemented wake-locks to monitor and control access to any hardware device that is either present or connected to the mobile phone device.

3. Provide virtualized clocks at virtual containers that adjust time scales relative to the underlying physical clock of the device, so that the rate of periodic tasks of an application inside a virtual container can be changed without the application's notice. This can be helpful in adjusting the behavior an application with limited rate adjustment options. To achieve that, we use a different scheduling algorithm on the processes that are marked in need of virtualized clock. These processes are treated either as having higher or lower priority that other processes that run in the Dalvik VM. For this, we leverage our

porting of the Linux Control Groups to create a new scheduling discipline. Moreover, we can modify the process scheduling algorithm in accordance with the hardware utilization from each process in order to achieve power saving for the entire system. While we can demonstrate 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 or policing or resources.

4. Provide a set of performance metrics to be used for the evaluation of our prototype and provide trade-off analyses of different approaches that pertain to both security and performance requirements.

2.5 Task 5: Prototype Evaluation on Android Platform

We focused our experimental efforts on Google's Android platform because it has an open development platform that facilitates the implementation of operating system modules and driver isolation. In addition, it is device independent, which means that we can develop our prototype on a desktop environment using QEMU [18] and then port it to the device of our choice.

1. We evaluated different virtualization technologies. In this sub-task, we evaluated different virtualization technologies and their limitations. This includes widely available virtualization technologies including VMWare, Xen, and OpenVZ.
2. We evaluated against malware instances.
3. We validated effectiveness of control mechanisms.
4. We evaluated performance metrics.
5. Video of malware exploits and defenses. We created videos showing how malware can exploit the platform and how our techniques can defend against. The video is generated using exploits for Camera and Microphone.

Activities in this task focused on quantifying early results from a proof-of-concept prototype in both performance and assurance.

3. Architecture and Technical Discussions

In this section, we provide architectural and technical details in our implementations of our proposed solutions.

3.1 Virtualization

To achieve the aforementioned system goals, we propose the following conceptual system architecture as shown in Figure 1.

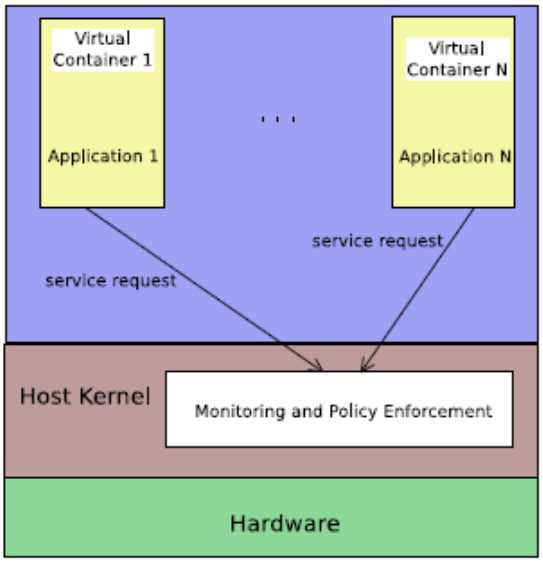

Figure 1: System Architecture

For commonly used applications 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
- System resource control
- Virtualized device support

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

3.1.1 File Level System Isolation

For file level system isolation, we use the built-in Linux *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.

As we know, *chroot* is an operation that changes the apparent disk root directory for the current running process and its children. This can be used for testing, recovery, privilege separation, and security. If run with root privileges, *chroot* can be broken however. If the goal of *chroot* is privilege separation and security, as in our case, then *chroot* should not be run as privileged user.

A *chroot* environment can be created either through some system provided tools or through a manual process. For our purposes, we need the following for our *chroot* environment:

1. A default shell
2. The *su* program to drop root user privileges
3. Basic tools (cp, ls, mv, etc)
4. Platform specific programs (dalvikvm, etc)
5. Supporting libraries for applications
6. Necessary directories (/tmp, cache directories, ...)
7. Necessary devices (framebuffer devices, etc)

One way to create a *chroot* directory is to use OS provided tools, if available, such as *debootstrap* for Debian:

```
$ debootstrap lenny /tmp/chroot
```

This will create a directory with common directories, library files, and commands. More commands can be added after the initialization.

Another way is to manually identify, copy, or create necessary files and directories. This is more flexible, but is also time-consuming. We use this approach for the Android platform since there is no ready-made system initialization commands such as "*debootstrap*" for the Android platform that we know of. We automate this process through shell scripts.

In the simplest form, a *chroot* operation may look like this:

```
$ sudo /usr/sbin/chroot /chroot/dir
```

This, however, starts the chroot environment with root privileges, so a drop of root privileges is necessary for safer operations:

```
$ sudo /usr/sbin/chroot /chroot/dir /bin/su - $USER -c "$PARAMETERS"
```

This will start the container with the privileges of the $USER, as defined by the user database file under the chroot directory.

In more general terms, a script should be used for initialization, validation, and privilege adjustment operations rather directly invoking the shell or the su command.

3.1.2 System Resource Control

Different mechanisms are used to support resource control depending on the method used to support system virtualization. These resource mechanisms are discussed below.

OpenVZ-based Resource Control. OpenVZ provides per-container resource accounting and limiting in four main areas:

- User beancounters
- Disk quota
- CPU scheduler
- I/O priorities for virtual environments and I/O accounting

User beancounters are a set of limits and guarantees for containers regarding system resources, such as memory, number of open files, etc. Disk quota controls how much disk space a container is allowed to use. CPU scheduler uses a two-level scheduling approach for choosing containers, and processes inside containers. I/O priorities assign different priorities for I/O operations for different containers.

CGroups Resource Control. Linux *control groups* (CGroups) are used as a mechanism to control a group of processes for specialized behavior. One advantage of CGroups compared to OpenVZ is that CGroups exists as part of the Linux kernel, and, unlike OpenVZ, is 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 are 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. Figure 2 demonstrates that effectiveness of CPU usage control on applications.

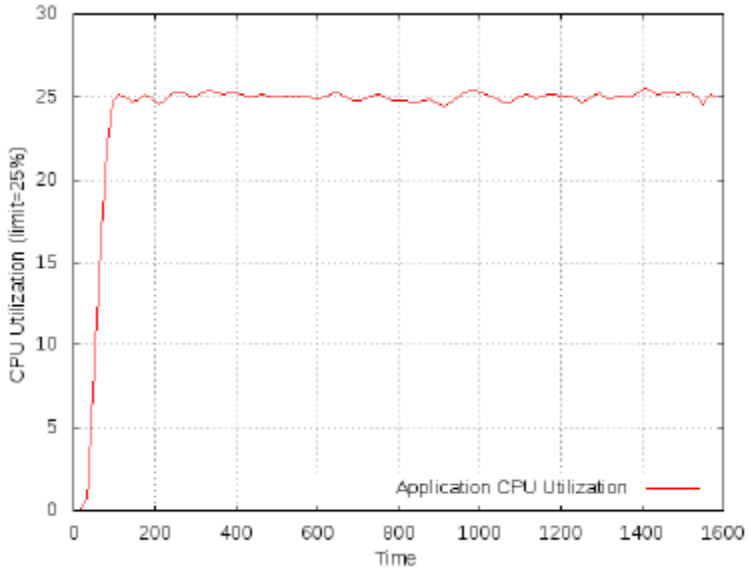

Figure 2: Application CPU Usage Control Using CGroups

Linux Built-in Control Mechanisms. 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.

3.1.3 Virtualized Device Support

The third area is the virtualized device support. In this work, we primarily focus on the virtualization of the display as a first step to provide interactive application support. For control groups approach, processes should allow access certain authorized devices only. This can be used in combination with a *chroot* environment to confine the process under a subdirectory, and mounting virtualized devices as needed under that subdirectory.

For the virtual container approach, each container will be assigned its own framebuffer device when started. Applications running inside a container can use the framebuffer device either directly, or through a third-party graphics library. Whenever a virtual container becomes active for user interaction, the host kernel will map the virtual container's framebuffer device to the native framebuffer so that the container's output will be displayed on the device screen. The host will then direct user data, including keyboard and mouse input, to the active virtual container to be processed. This enables fast switching between different applications running inside separate containers. A conceptual illustration is shown in Figure 3.

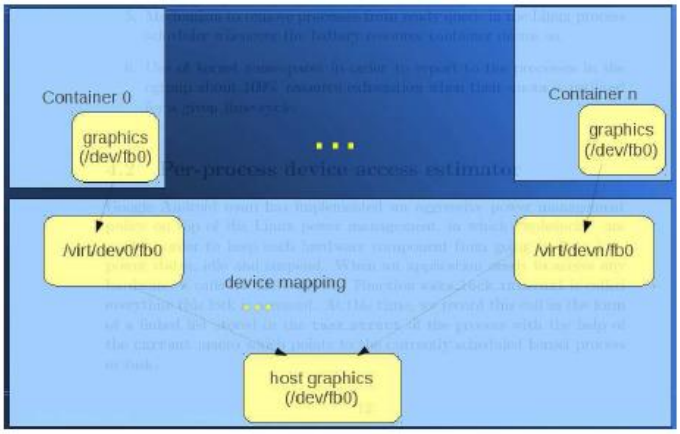

Figure 3: Virtualized Display Architecture

For faster application startup times inside virtual containers, we use pre-created virtual containers when starting applications. In this scheme, the host operating system tries to make sure that there will be a pristine virtual execution environment ready when an application is started by creating spare virtual containers, and when pre-created containers are about to be used up. By combining quick startup and fast switching between applications, users are provided with a responsive execution environment for interactive applications. The framebuffer device and virtual display implementation are further discussed below.

Android Framebuffer Device. Android relies on the standard framebuffer device (/dev/fb0 or /dev/graphics/fb0) and driver as described in the linux/fb.h kernel header file, which provides an abstraction for the graphics hardware. Framebuffer device represents the framebuffer of some video hardware and allows application software to access the graphics hardware through a well-defined interface.

Android uses two buffers for the display device: *front buffer* and *back buffer*. The front buffer is used for composition and the back buffer is used for drawing. Android flips these buffers by adjusting buffer offsets to ensure minimal data copying between these two buffers. Because of this design, Android requires a linear address space of mappable memory twice the size of hardware display (in pixels).

A typical frame display procedure is as follows:

- Accessing the driver by opening `/dev/fb0` or `/dev/graphics/fb0`.
- Using `ioctl` calls to retrieve display information. The request types are `FBIOGET_FSCREENINFO` or `FBIOGET_VSCREENINFO`.
- Using `FBIOGET_VSCREENINFO` to map needed space and set pixel format, such as `rgb_565`, etc.

Virtual Display Implementation. The framebuffer driver is normally implemented as a character device under the Linux kernel. For display virtualization, we write a kernel module that implements the framebuffer operations. For debugging and testing purposes, we use a regular Linux installation (Debian) besides the Android platform. For Android platform, we implement the framebuffer driver based on the sample driver implementation (pguidefb.c), and for the regular Linux distribution, we base our implementation on drivers/video/vfb.c.

To create multiple framebuffer devices, we insert multiple modules that only differ in driver name (so that more than one driver can be inserted). The driver name can be defined in the platform_driver structure as follows:

```
static struct platform_driver vfb_driver = {
        .probe  = vfb_probe,
        .remove = vfb_remove,
        .driver = {
                .name   = "vfb1",
        },
};
```

Assuming there are no framebuffer devices (under /dev), the following commands can be used to load modules to create two framebuffer devices under our regular Linux system:

```
$ modprobe vfb
$ insmod vfb1.ko
$ insmod vfb2.ko
```

After this step, two framebuffer devices, /dev/fb0 and /dev/fb1 should appear. We also implemented helper programs to modify framebuffer device settings after the devices are installed.

The function pointed ".probe" member of the structure above will be called at the initialization stage to register framebuffer device, and to set up variables, memory space, etc.

More operations that the framebuffer device should support are specified using fb_ops structure. A sample declaration is given as follows (from Android's pguidefb.c):

```
static struct fb_ops pguide_fb_ops = {
    ...

    .fb_check_var   = pguide_fb_check_var,
    .fb_set_par     = pguide_fb_set_par,
    .fb_setcolreg   = pguide_fb_setcolreg,
    .fb_pan_display = pguide_fb_pan_display,

    ...
};
```

The .fb_check_var function should be implemented to check if a variable is supported by the hardware device, and .fb_set_var function should be able to set video settings, including rotation, etc. .fb_setcolreg is used to software color map. As discussed above, when a page flip is required, Android makes a call FBIOPUT_VSCREENINFO ioctl call with a new y-offset pointing to the other buffer in video memory. The .fb_pan_display function is used in order to do the actual flip.

After allocating the video memory and setting the pixel format, the system can use mmap() to map the memory into the process's address space. All writes to the framebuffer are done through this mmaped memory.

After the framebuffer devices are initialized, they should be used in the virtualization context as depicted in Figure 3. We use the following steps to implement the device virtualization and isolation:

- Create/install framebuffer devices at the host system.
- Create a directory for each container to be mounted inside chroot directory.
- Put links for framebuffer devices under appropriate directories created above.
- Enter the chroot environment:
    - Enter container (as root)
    - Create a regular user
    - Set permissions to the regular user to access virtual devices
    - Drop privileges to become the regular user and execute requested command

Below are sample scenarios for the steps described above:

First we create framebuffer devices:

```
$ ls /dev/fb*
$ ls: cannot access /dev/fb*: No such file or directory
$
$ modprobe vfb
$ insmod vfb1.ko
$ insmod vfb2.ko
$
```

```
$ ls /dev/fb*
$ /dev/fb0  /dev/fb1
```

Then we create directories to be mounted under chroot directories, put links to framebuffer devices, and mount them under chroot directories:

```
$ mkdir /dev/virt0
$ mkdir /dev/virt1
$
$ ln /dev/fb0 /dev/virt0/fb0
$ ln /dev/fb1 /dev/virt1/fb0
$
$ mount --bind /dev/virt0 /tmp/chroot0/dev/graphics
$ mount --bind /dev/virt1 /tmp/chroot1/dev/graphics
```

Now we setup the environment by fixing file names and adjusting permissions:

```
$ chroot /tmp/chroot0 /bin/ln -s /dev/graphics/fb0 /dev/fb0
$ chroot /tmp/chroot0 /usr/sbin/useradd android
$ chroot /tmp/chroot0 /bin/chown android /dev/fb0
$ chroot /tmp/chroot0 /bin/su - android UserCommand -c "Parameters"
```

Similar steps can be done for other chroot environments. This way, the user android will only be able to execute the command UserCommand with Parameters inside the /tmp/chroot0 directory. The user can modify /dev/fb0 inside the new container, which is mapped to host's /dev/fb0. However, it cannot access other files or devices from inside the container.

3.2 Resource Management

3.2.1 Linux Control Groups

Control Groups provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behavior. One main advantage of *cgroups* mechanism is that it already exists in the Linux kernel, and can readily be used. Another advantage is it is very low overhead and provides dynamic control available through simplified file operations. Linux CGroups provide facilities to control CPU, memory, network, etc, as follows:

- Cpuset
- Cpu
- Cpuacct
- Memory
- Devices
- Freezer
- Net_cls

The CPU subsystem provides mechanisms to control CPU usage of processes. The CPUSet subsystem assigns processes to logical CPUs as seen by the operating system. The CPUAcct subsystem provides CPU usage statistics for processes. The memory subsystem controls memory usage and provides related statistics, such as the number of times maximum memory limit has exceeded. Devices subsystem controls access to system devices. Freezer provides an easy way of putting processes to sleep or waking them up. Net_cls provides a method for marking network packets originating from processes in a specific control group so that further processing can be done at the OS level using tools such traffic control (tc) or iptables.

Most of such implementation efforts are geared towards traditional computing platforms, and do not have support for battery management, an important resource for hand-held mobile devices. In this project, we explored ways to utilize and augment currently available CGroups mechanisms listed above. Besides built-in control mechanisms, we also extended CGroups to support battery usage accounting through a new battery management subsystem. Shown in Figure 4 is a diagram of our Android implementation with kernel patches (shaded) to the subsystem components.

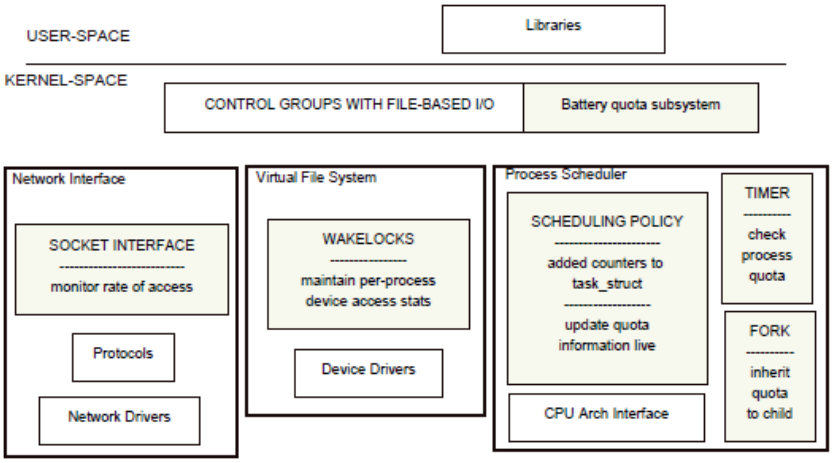

Figure 4: Android Implementation Diagram

3.2.2 Battery Basic Container Design

Our Battery Resource Container needs the following components:

1. Per-process device access estimator.
2. Live calculation of device or hardware component's relative weights for battery charge consumption.
3. Live per-process battery charge calculator based on our energy model.
4. Container using the cgroups infrastructure to group processes together based on any defined criteria.
5. Mechanism to remove processes from ready queue in the Linux process scheduler whenever the battery resource container deems so.
6. Use of kernel namespaces in order to report to the processes in the cgroup about 100% resource exhaustion when their quota is utilized for a given time cycle.

Top-level modules associated with Battery Resource Container are shown in Figure 5.

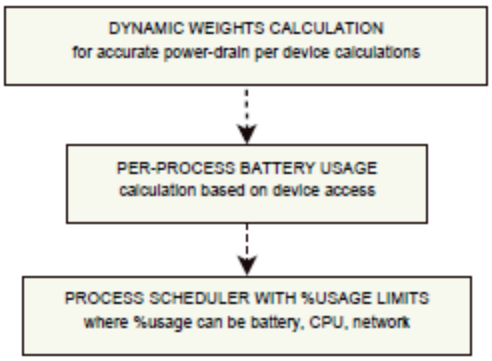

Figure 5: Top-Level Modules

3.2.3 Per-Process Device Access Estimator

Google Android team implemented an aggressive power management policy on top of the Linux power management, in which "wakelocks" are used in order to keep each hardware component from going back to low-power states, idle and suspend. When an application needs to access any hardware, it calls for the wakelock. Function wake_lock_internal is called every time this lock is accessed. At this time, we record this call in the form of a linked list stored in the task_struct of the process with the help of the current macro which points to the currently scheduled kernel process or task.

It is however not sufficient to know which process accessed which hardware component. It is also important to know the level of resource consumption. It is important to know the frequency at which the CPU operated to service a given task, and also, the amount of data that was actually transferred over the network by each process.

In order to account for the actual amount of data sent over the network by a process we watch for all packets in the kernel sockets and attribute a packet created to the current process which has been scheduled by the kernel. This is done when sock->ops->sendmsg(iocb, sock, msg, size) function is called. The size is added to the previously recorded number of bytes of data sent over the network by the process associated with the current macro.

For all other hardware components we assume that within the small kernel time cycle, only one single process accesses the hardware in question, at full consumption level. This is especially true for display, where the entire screen is taken over by a single view which in most cases is a single process; although the varying levels of brightness do cause variations in battery consumption, which must be accounted for, on our later models.

The accuracy of our model for estimating the device access at runtime on a per-process level is debatable. This is an area that can be further researched. We do not have alternate implementations at hand, at the moment, to verify our usage estimations against.

3.2.4 Energy Model

The calculations are based on the actual milliampere-hour (mAh) battery charge reported by the battery driver to the kernel. Since we watch this value during every timer cycle in-kernel, we do not need to separately implement our own sampling window. Therefore, a simple formula can be used to calculate the live battery charge per-process:

$$\triangle B_P = \sum_H (\triangle B_H * (\triangle H_P / \triangle H)) \omega_H$$

B=battery; P=process; H=hardware component

Battery charge per-process is the weighted sum of battery charge consumed by each device during current kernel cycle multiplied by the percentage usage of that device by the same process. Google Android has the weights pre-calculated and approximated for simplicity. These weights are stored in an XML file in their source tree. While we can start the calculations using these weights, it is highly inaccurate to use these same weights for every handset.

These weights can be adjusted in the live system using a simple stochastic model. The devices or hardware components in question here are *CPU, display, WiFi, radio, audio, GPS,* and *Bluetooth.* This can be engineered by comparing the current rate of fall of battery charge and the list of devices which are active in current time cycle with the summary of previous results.

3.2.5 CGroups Based Subsystem for Battery Quota

Control Group is 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 different policies either decided by the user, or the system. On Android, if we group together tasks based on their userid, we can set quotas for individual application. The policies that are to be used have been left out of the scope of this work.

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 and we go forward to implement our battery quota subsystem that can be enforced upon this process set in the cgroup. The underlying mechanism consists of three elements: the control groups, subsystems, and hierarchies. Our *battery* quota subsystem has an interface that allows for assigning percentage of usage limits over the process set in the cgroup associated. The file system has the following elements:

`battery.quota.percent`. This is used to set the percentage limit to battery consumption. If the tasks in the current cgroup end up using the given amount of percentage battery, 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.charge`. This is used to set the absolute battery charge limit to battery consumption.

`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 are not allowed to exhaust the battery, unless there are others in the system having an equal piece of it.

`battery.next_balance`. Some interface to determine the number of jiffies until the current task is ready to be scheduled again.

3.3 User-Space Library to Control the Setup of Containers

The cgroup framework needs to be initialized from user-space after every reboot. Regular file 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:

mount –t cgroup –o cpuset,battery taskset1 -/cgroup/taskset1

Here, two subsystems, cpuset and battery 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 tasks in the cgroup/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 needs to be maintained across reboots, care has to be taken to save this information in user-space.

On Linux, there is a library libcgroup and other tools like the cgred daemon, which do not yet exist in the Google Android's world. These user-space functions can be created as a Dalvik service, or as an Android OS's system component. Decisions will have to be made, and a design proposal is required for implementing these user-space support tools. However, they are not required, in order to test the working implementation of our Battery Resource Containers and hence this task has been left out of the scope of this study.

4. Conclusions and Recommendations

This project explored security architectures for mobile computing devices. With the increase of computing power, memory, storage and wireless connectivity, mobile wireless devices are being increasingly used by a wide spectrum of users, in everyday life as well as in military scenarios. These new hand-held devices are capable of carrying significant amount of personal data, like financial bank-related information and important documents, while also used in mission-critical operations.

Most hand-held mobile 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 increased 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, 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. As a result, users and missions are placed at risk from both malicious apps as well as attacks against vulnerable apps.

The goal of this project was to develop secure architectures for securely running apps on smart phones. We developed a light-weight, containment-based security architecture for mobile wireless devices to provide high assurance of security from malicious third party apps. The framework we developed enables the mobile phone operating system to monitor, profile, control, and enforce policies on applications. We employ process isolation to confine program execution in well-defined virtual containers to block any unauthorized access and modification of files and other system resources. We also implement resource management, to regulate resource usage and provisioning for processes or process groups. We believe our architecture and prototype demonstrated that the goal of ultimately providing efficient and effective security mechanisms to support a wide variety of security policies against malicious apps for mobile devices is achievable.

Our preliminary work in this seedling effort demonstrated effective confinement of malicious apps on Android platforms. Recommend further investment to mature this technology.

5. References

[1] 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.*, 14(6):745–768, 2008.

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

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven H, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *In SOSP*, pages 164–177, 2003.

[4] KVM (Kernel based Virtual Machine). http://kvm.qumranet.com.

[5] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, Oct. 2000.

[6] Google. Android open source platform: http://www.android.com.

[7] Tom Goovaerts, Bart De Win, Bart De Decker, and Wouter Joosen. *Assessment of Palm OS Susceptibility to Malicious Code Threats*. 2005.

[8] Yih Huang, Angelos Stavrou, Anup K. Ghosh, and Sushil Jajodia. Efficiently tracking application interactions using lightweight virtualization 1, 2008.

[9] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *In Proceedings of the ACM Conference on Computer and Communications Security (ccs)*, 2007.

[10] Niu Limin, Tan Xiaobin, and Yin Baoqun. Estimation of system power consumption on mobile computing devices. *Computational Intelligence and Security, International Conference on*, 0:1058–1061, 2007.

[11] Benjamin R. Moyers, John P. Dunning, Randolph C. Marchany, and Joseph G. Tront. Effects of wi-fi and bluetooth battery exhaustion attacks on mobile devices. *Hawaii International Conference on System Sciences*, 0:1–9, 2010.

[12] Daniel C. Nash, Thomas L. Martin, Dong S. Ha, and Michael S. Hsiao. Towards an intrusion detection system for battery exhaustion attacks on mobile computing devices. *Pervasive Computing and Communications Workshops, IEEE International Conference on*, 0:141–145, 2005.

[13] Venkat Rao, Gaurav Singhal, Anshul Kumar, and Nicolas 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*, pages 105–110, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Chiyoung Seo, Sam Malek, and Nenad 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*, pages 421–424, New York, NY, USA, 2007. ACM.

[15] VirtualBox. http://virtualbox.org.

[16] Virtuozzo. OpenVZ virtualization: http://www.openvz.org.

[17] VMware. http://www.vmware.com.

[18] QEMU. http://www.qemu.org.

List of Symbols, Abbreviations, and Acronyms

AFRL	Air Force Research Laboratory
API	Application Programming Interface
ARM	Advanced RISC Processor
BSD	Berkley Software Distribution
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency
EDGE	Enhanced Data rates for GSM Evolution
GMU	George Mason Universit
GPS	Global Positioning System
GSM	Global System for Mobile Communications, originally Groupe Spécial Mobile
IDE	Integrated Development Environment
I/O	Input/Output
KVM	Kernel-based Virtual Machine
mAh	milliampere-hour
OpenVZ	Open Virtualization
OS	Operating System
PV	Paravitualization
QEMU	Open source processor emulator
SDK	Software Development Kit
UML	Unified Modeling Language
USB	Universal Serial Bus
VM	Virtual Machine
WiFi	Wireless Fidelity (IEEE 802.11b wireless networking)
Xen	Virtual machine monitor for IA-32, X-86-64, Itanium, and ARM architectures