L4Android: A Generic Operating System Framework for Secure Smartphones

Matthias Lange[1] Steffen Liebergeld[1]
Adam Lackorzynski[2] Alexander Warg[2] Michael Peter[1]

[1]Security in Telecommunications
Technische Universität Berlin and Deutsche Telekom Laboratories
{mlange,steffen,peter}@sec.t-labs.tu-berlin.de

[2]Chair of Operating Systems
Technische Universität Dresden
{adam,warg}@os.inf.tu-dresden.de

ABSTRACT

Smartphones became many people's primary means of communication. Emerging applications such as Near Field Communication require new levels of security that cannot be enforced by current smartphone operating systems. Therefore vendors resort to hardware extensions that have limitations in flexibility and increase the bill of materials. In this work we present a generic operating system framework that does away with the need for such hardware extensions. We encapsulate the original smartphone operating system in a virtual machine. Our framework allows for highly secure applications to run side-by-side with the virtual machine. It is based on a state-of-the-art microkernel that ensures isolation between the virtual machine and secure applications. We evaluate our framework by sketching how it can be used to solve four problems in current smartphone security.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General – System architectures

General Terms

Security

Keywords

Smartphones, Secure Operating Systems, System Virtualization, Near Field Communication

1. INTRODUCTION

Smartphones have become omnipresent devices. They combine the computing power previously known from desktop computers with the mobility and connectivity of cellular phones. With their plethora of interfaces like Bluetooth, Wifi, and the cellular network they remain connected to the Internet at all times. User-installable applications allow users to customize their smartphone with features according to their needs. Therefore, many people have adopted smartphones as their primary means of communication.

With their computing power and widespread adoption, smartphones promise to materialize the idea of ubiquitous computing. That made major companies such as Google, Apple, and Microsoft invest a lot of money and resources to increase the rate of innovation. New applications and use cases, which were previously unimaginable on this class of devices, appear at a short rate. One emerging trend is using Near Field Communication (NFC) for authentication, mobile payment [24], ticketing and secure banking. These applications have high demands on security.

It is the task of the operating system (OS) to enforce the security of the system. Android has become the most popular mobile OS [13] in terms of market share. It is developed by Google and the Open Handset Alliance as an open source project. Unfortunately Android is riddled with security problems, such as delayed security updates and an insufficient access control model [15]. To improve on Android's security, a number of solutions were proposed by the academic research community. These range from extensive taint tracking [46], behavioral analysis [33], mocking interfaces [7], application of mandatory access control [47, 17], analysis of remote duplicates [22], label based tracking [36], to an implementation of a custom *privacy mode* [49].

These solutions rely on the integrity of the Android kernel. However, kernel integrity cannot be ensured when a device is rooted. Rooting can happen voluntarily by the user to enable custom features or by malware due to a vulnerability of the Android kernel. A recent study found 88 vulnerabilities in the Android kernel [14], which shows that unintended rooting is a serious threat. This number of vulnerabilities is inherent to the monolithic architecture of the Android kernel. Therefore Android is not suited for prospective applications that have high demands on security.

We address future demands on security with a novel OS base architecture. In our work, we do not improve on Android security, but provide an isolated environment for applications with high demands on security. The Android OS is securely encapsulated inside a virtual machine (VM).

1.1 Contributions

This work has two major contributions. One is the introduction of a general OS framework for mobile devices that allows for highly secure applications. The second contribution is the detailed evaluation of our framework. There we

SPSM'11, October 17, 2011, Chicago, Illinois, USA.

present how our framework can solve current smartphone security issues.

We designed a generic framework for secure smartphone architectures and implemented a proof-of-concept setup that runs on a real phone. It consists of three core components. A state-of-the-art microkernel establishes high assurance isolation boundaries. Our framework comprises the infrastructure for applications with high demands on security. These are implemented as applications directly on the microkernel. Consequently, these applications have a tiny trusted computing base (TCB) and lend themselves to certification. The third core component are virtual machines, which act as a secure container to run the original smartphone OS.

We show how our framework can be used to solve four problems in current smartphone security.

Software Smartcards: Our framework facilitates the secure implementation of smartcard functionality in software.

Unified Corporate and Private Phone: We show how a private and a business phone can be unified on one device in a secure manner.

Mobile Rootkit Detection: We describe how our framework enables rootkit detection on mobile devices.

Hardware Abstraction: We propose to implement device specific drivers in a layer below Android. This allows Google to supply generic kernel versions that are readily applicable to all devices, which allows for much faster security updates. It also enables vendors to supply non GPL compliant drivers without violating Linux' development rules.

The rest of the paper is organized as follows. In Section 2, we give an overview on the background of our work. Section 3 describes details on the vulnerabilities of Android. We proceed in Section 4 with details on the design of our framework. L^{4}Android is our prototype implementation of the framework which we describe in Section 5. In Section 6, we demonstrate how our framework solves four long-standing security challenges. Section 7 gives an overview on related work. In Section 8 we conclude and summarize our results and contributions. We also provide an outlook on future work.

2. BACKGROUND

In this section we provide the reader with background information on microkernels and virtualization in the context of mobile devices. We briefly outline the general hardware architecture of a smartphone. The architecture of Android is described in Section 2.4.

2.1 Microkernel

The OS kernel runs at the most privileged mode of the CPU (*kernel* or *supervisor mode*). The applications run with less privileges in *user mode*.

Because of the kernel running with the highest privileges, all functionality critical to maintain system security has to be implemented in kernel mode. The essential mechanisms implemented in the kernel are protection domains, scheduling and means of communication between protection domains.

Additionally to those essential mechnisms, monolithic kernels such as Linux implement more functionality like file systems, device drivers, and protocol stacks. The drawback of this architecture is that monolithic kernels contain functionality with high complexity. Because no means of isolation exist in kernel mode, any in-kernel bug can modify kernel memory, leading to crashes or to the installation of rootkits.

In contrast to monolithic kernels a microkernel implements only the essential mechanisms outlined above. File systems, device drivers, and protocol stacks are implemented as user-mode tasks [27]. This dramatically reduces the complexity of the kernel. Isolation between user-mode tasks is enforced with address spaces. All communication between tasks is done via efficient explicit kernel-mediated inter-process communication (IPC).

2.2 Virtualization

A formal definition of virtualization was made by Popek and Goldberg [21]. They define a VM to be an *efficient, isolated duplicate* of a real machine. A special piece of software, the *virtual machine monitor* (VMM), establishes the VM. The formal model requires a CPU with two modes, a privileged mode (kernel mode) and a less privileged mode (user mode). Guest code runs in user mode under the control of the VMM.

For an instruction set architecture (ISA) to be virtualizable all *sensitive* instructions need to trap into kernel mode when executed in user mode. Sensitive instructions are instructions that produce different results when executed in user mode as compared to being executed in kernel mode. According to this definition the x86 [28] and ARM [40] ISA were found to be non-virtualizable.

To implement virtualization on non-virtualizable architectures, several well-known workarounds exist.

Emulation In an emulator all guest-instructions are interpreted and their results computed in software. Hybrid approaches allow for guest user-mode code to execute directly on the host CPU. However, emulation comes with a significant performance penalty [20].

Binary Translation In systems using binary translation, the guest-code is modified prior to execution. Sensitive instructions are replaced with either non-sensitive ones or with explicit calls to the VMM [30]. Binary translation can be more efficient than pure emulation, but is still less efficient than native execution.

Rehosting OS rehosting is a port of the guest's kernel from the machine interface to the host's kernel interface. This effectively makes the guest kernel an application of the host OS. Such a port does not require an additional VMM[1].

To enable more efficient virtualization solutions, Intel and AMD added hardware virtualization capabilities to their CPUs [42]. ARM recently announced similar virtualization support for their Cortex-A15 design [41]. While hardware virtualization support is widespread in desktop systems today, it hasn't found its way into the embedded market yet.

[1]However, one can argue that the adapted OS-kernel takes the role of the VMM.

2.3 Smartphone Architecture

Modern smartphones comprise many complex subsystems. The central one is the application processor that runs the smartphone OS such as Android or iOS and all the applications. Other systems include the baseband, GPS and audio hardware. See Figure 1 for a conceptual hardware layout of a smartphone.

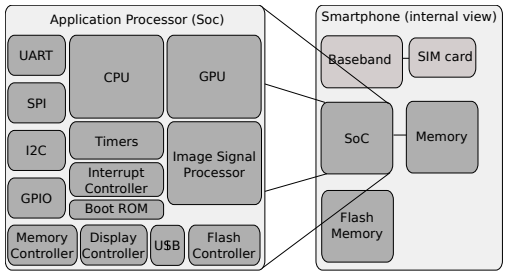

Figure 1: The basic design of a modern smartphone.

The application processor comes in the form of a System on a Chip (SoC). The CPU, memory and interrupt controller, timer as well as additional functional units are integrated on one silicon chip. This design helps to reduce the bill of materials and power usage. Usually the SoC also integrates a graphics processing unit and controllers for system buses such as I2C, SPI and USB. These buses are used to connect peripheral devices.

The baseband is the phone's gateway to the cellular network. The baseband uses credentials stored on the SIM card to identify a subscriber on the mobile network. The SIM card is a smartcard and forms an isolated execution environment.

2.4 Android Architecture

In general Android is a software stack for smartphones and tablets. It consists of the kernel, the Android runtime, libraries, an application framework, and the applications. Each of these parts will be described briefly in this section.

Kernel Android is based on a specially crafted Linux kernel. Google enhanced Linux to better address the needs of mobile platforms with improved power management, better handling of limited system resources and a special IPC mechanism.

Libraries Android provides a set of native libraries that are used by various components in the system. This includes libraries for media, 2D/3D graphics, and a custom C standard library (bionic). The functionality of these libraries is exposed to applications by the Application Framework. Many libraries are based on open source projects. Examples are WebKit and SQLite.

The **Android Runtime** is mainly made of the Dalvik VM, a register-based Java virtual machine. Dalvik runs Java code compiled to a special format (dex), which is optimized for low memory footprint. Everything on top of this layer is written in Java.

Applications Android applications are written in Java. Android ships with a set of core applications for telephony, personal information management, and Internet browsing. For improved performance, applications can include native code written in the C language. Native code is integrated with the Java code through JNI. It does not benefit from the Java abstractions (automated memory management, garbage collection).

3. THREATS

We identified four main issues which make Android vulnerable to attacks. First, security critical software updates are delayed or not deployed at all. Second, the Linux kernel is not an adequate OS kernel for secure systems. Third, rooted phones disable many of the security features, which are in place. Fourth, Android's permission system is too coarse grained to effectively protect critical system resources.

3.1 Delayed System Updates

Android is an open source software project. In addition to the custom components implemented by Google, it builds on other open source projects such as WebKit and the Linux kernel. It is characteristic to such projects that they are developed by a loosely coupled international community. To keep track of the development process they use publicly available source code repositories and public bug trackers to collect bug reports.

In software security the time span from the discovery of a vulnerability until the deployment of the security patch is critical. During this time span the system is vulnerable and attackers race to create exploits. Vulnerabilities become known to the general public as soon as the resulting patch is submitted to the public repository. This increases the visibility of a vulnerability for attackers, who may start to create exploits and attack unpatched systems. Therefore timely patch deployment is vital for a system's security. In Linux distributions, for example, lots of effort is spent on update systems to ensure quick deployment of security updates.

To set themselves apart from others, device manufacturers augment Android with custom user interfaces and features. These additions require in-depth modification of the Android source code. Google largely develops Android behind closed doors and releases the source only at certain milestones. This style of development has the drawback that the device vendors cannot continuously keep their source tree up to date. Therefore they need to port their custom user interfaces and features to the new version as soon as the new code is released by Google. After porting, the features need to go trough quality management. All in all, this is a time consuming and costly effort, which is often prohibitive. Keeping their software up-to-date is of little value for the device manufacturer in terms of market value, when the device is already sold.

Android does not allow for selective updates, but relies on full system images that have to be provided by the device manufacturer. A full system image requires bandwidth and disrupts the user's work flow. Therefore, device manufacturers usually do not fix individual vulnerabilities but accumulate updates.

These factors introduce a significant delay in the deployment of updates, which results in millions of devices with known and unpatched vulnerabilities.

Security features, such as full disk encryption, are introduced with new Android releases, but are not backported to existing versions. A recent study by Google [23] about the distribution of different Android versions revealed that more than 90% of the Android devices are still using Android 2.1 and 2.2. The most recent version 2.3 released in Decem-

ber 2010 is deployed on less than five percent of Android smartphones. This shows that Google's attempts at introducing better security has limited effects for devices that are already deployed.

3.2 Linux Kernel

Android is based on the Linux kernel. Linux implements a monolithic architecture. All kernel components, including device drivers, run in kernel mode, where no isolation between components is provided. Any kernel bug that can be exploited enables an attacker to modify kernel memory, and thereby mitigate all security measures of the kernel. Therefore kernel updates, as well as extensive testing and validation of kernel code are vital to Android security.

Device manufacturers often need to implement custom drivers for their hardware. This driver code is often not contributed back to the Linux community. As a consequence these drivers do not go through the community review process and are often of poor quality. Porting the drivers to new versions of Linux is often not considered worth the effort due to cost constraints and the work required for test and validation. Therefore many devices run outdated Linux kernels.

A recent study on the stock Android Froyo kernel, version 2.6.32, found 88 security critical bugs [14]. This gives a rough impression of the security of Android kernels. However, due to the bad driver code supplied by vendors, we suspect the error rate of deployed kernels to be even higher.

3.3 Rooted Phones

Rooting is the process that overcomes the kernel's integrity barrier. It can happen in two ways. First, voluntarily by the user who wants to be able to install additional, potentially unauthorized applications. This type of rooting is often done by installing a modified firmware, including a new kernel image, on the device. Second, by malware such as DroidDream [1] in order to gain maximum privileges on the infected system. This type of rooting is achieved by exploiting known security flaws in the respective smartphone OS.

Rooting requires tampering with the OS kernel which destroys its integrity. A rooted system cannot put trust in its kernel. The modified kernel might disable Android security measures, contain malware such as key loggers, or subtly alter the system's behavior to leak private information. Therefore rooting is a serious threat to smartphone security. This problem becomes even more pronounced, since two major Android device vendors announced rootability as a marketing feature [39, 18] for their devices.

3.4 Android Permission System

Android implements mandatory access control (MAC) in the form of a permission system. At installation time an application can request permission to access system resources such as location, Internet, or the cellular network, from the user. The user is then presented with a screen allowing him to either grant all the permissions or cancel the installation. It is not possible to selectively accept or deny access privileges. Thus, many users simply accept such permission requests without considering their implications.

Another problem is that the permissions are too coarse grained [15]. If an application was granted Internet access, it is free to communicate with any server on the Internet. If this application was also granted access to the Android address book, nothing prevents it from sending the address book's content to a remote server.

Using live taint tracking, Enck et al. [46] found that two thirds of the applications they analyzed, exhibited suspicious handling of private data. Static code analysis revealed potential privacy leaks in even more applications [45].

With many applications being distributed via the Android market, its acceptance process has the potential to filter malicious applications. However, the Android market was found to distribute malware [16, 31].

4. FRAMEWORK

We consider the monolithic architecture of Android as the main reason for its security problems. Monolithic components consist of numerous subsystems. A monolithic component needs to be equipped with all permissions required by its subsystems. A bug in one of them suffices for an attacker to tamper with any part of the component and to leverage all of its permissions.

The design of our OS frameworks is based on the principle of divide and conquer. Instead of having complex monolithic components, the framework hosts multiple smaller components. Each component implements one basic service and is equipped with only the permissions needed for its correct operation (Principle Of Least Authority, POLA [34]). This helps confining attacks to the concerned component.

High level functionality is implemented with the help of many basic components that communicate with each other. Thereby the system behaves like a distributed system. Thus, we need a secure communication mechanism that enables components to request services from one another. Communication requires a naming mechanism to locate a communication partner, and a mechanism to enforce access permissions.

Dividing monolithic systems into smaller subsystems is a complex task, because these subsystems have complex dependencies with one another. This problem is prominent with OS kernels. Therefore it is not possible to apply our OS construction mechanism to existing OSes. Instead, our framework provides virtual machines to run existing systems. Security conscious applications are implemented outside of the VM. Even in the event of a compromised VM these applications maintain their integrity.

In the next sections we will describe the basic building blocks of our system. We start with an in-depth description of how components are isolated, and how we handle secure communication. Section 4.2 will introduce the microkernel, which forms the secure anchor of our framework. Furthermore we provide the reader with details on our concrete implementation. Our microkernel is augmented with a runtime environment that implements basic services for applications, and thus facilitates the design and development of secure applications. We will then proceed to describe how our framework supports virtual machines to host existing smartphone OSes.

4.1 Component Isolation and Interaction

Our system encapsulates subsystems in *components*. Components are implemented in *protection domains*, and it is the duty of the OS kernel to ensure temporal and spacial isolation between components. Components can provide *services*

to other components. Services are modeled as *objects*, and are implemented inside protection domains.

Components can request services from other components via explicit message passing. To avoid the problem of the confused deputy, we went for *object-capabilities* [6], which unite naming and access permissions. Holding a capability entitles a component to communicate with the respective service. The kernel enforces the access permissions of capabilities.

We resorted to a naming scheme, where capability names are valid inside the respective component only (local naming). This greatly facilitates nesting of subsystems.

The capability-based communication system forms the basis for any other functionality in the system. In the following sections we will show how it is used to build the functionality for a whole system.

4.2 Secure and Small Kernel

The kernel is the only component that is running with the highest system privileges. It is responsible for establishing and maintaining the isolation of components running on top as applications.

Because of the characteristics outlined in Section 2.1 a microkernel is a suitable foundation for our OS framework. It has a considerably lower complexity than a monolithic kernel. This benefits applications in that their TCB can be optimized for their use-case.

The aforementioned concepts are implemented in a modern microkernel named *Fiasco.OC*. It is being developed as an open source project, and its sources are available at its public website[2].

4.2.1 Kernel Functionality

To maintain isolation and functionality of the system, the kernel offers the following key mechanisms.

Protection Domains To establish isolation between components the kernel provides protection domains, called *tasks*. Each task implements an address space for providing virtual memory and a capability space holding the capabilities for that task.

Execution is provided by *threads*. A thread executes the code provided by the applications. A task can host multiple threads, facilitating multi-processing environments.

Communication plays a crucial role in a system consisting of many components that need to interact with each other. Message passing, also called IPC, is an operation where two threads explicitly invoke a message passing operation to be able to exchange a limited amount of payload data. IPC is a synchronous operation, meaning that a thread will block until the communication partner has also entered the corresponding IPC operation.

Interrupts are a mechanism to handle asynchronous events. They are used for both hardware device-generated interrupts and for software-generated events. The receiver of an interrupt cannot decide whether the event has been generated by hardware or by another software component. When used as a software mechanism, interrupts allow for asynchronous notifications between threads, where one thread triggers an interrupt and continues execution. The receiving thread receives the event when it is ready. No payload can be transferred with this operation. Interrupts are modeled using IPC.

Scheduling is implemented in the kernel and is responsible for dispatching a thread that is ready to run. The default scheduling scheme is a fixed-priority, round-robin scheduler that strictly selects the thread with the highest priority. Within the same priority level the threads are scheduled round robin. The scheduler is also used to place threads on the different cores in a multi-core system.

Virtualization Containers When using hardware assisted virtualization[3] the kernel must implement the basic functionality of creating virtual machines and implement world switching. The object implementing this virtualization container is called *VM*. A VM is similar to a task and provides the guest physical memory.

Threads, typically those implementing a virtual machine monitor, have to use special functionality to enter a virtual machine for doing a 'world-switch' and to start executing guest code.

Object Creation *Factory* objects are used to create new objects in the system. The concept does not only cover kernel objects but all objects that may exist in the system. For the kernel, the kernel-factory can, for example, create new tasks, threads, and interrupts. Factories also implement a resource accounting scheme to prevent resource exhaustion.

4.2.2 Resource Access Delegation

Having described the functionality of the kernel still leaves open how access rights to resources are transferred between protection domains. Those resource can either be memory pages or objects.

The corresponding operation is called *map* and can be used in two variants. The first version is available when a task holds a capability on another task and thus can grant access rights to it. Using this type of operation does not require any consent of the receiving task. In contrast, the second type uses specifically typed IPC messages that allow transfering access rights to a receiver. The receiver specifies where those resources should be mapped to in its virtual memory or capability space.

The map operation implements the principle of privilege attenuation (POPA) [34]. A sender of a mapping can only grant access rights to others that it possesses itself, with the same or lower rights. For example, a memory page that is writable for the sender can be mapped with read-only rights to a receiver. A read-only mapping can only be mapped to other tasks with read-only permission.

The reverse operation of map is called *unmap*. This operation recursively revokes access rights from those protection domains that had previously received the access rights.

4.2.3 Fault Handling and Resolution

Fault and exception handling is implemented in user-mode services. Upon receiving a fault, the microkernel forwards

[2] http://os.inf.tu-dresden.de/fiasco/

[3] Examples being Intel VT and AMD's SVM

those to a designated handler thread. Such a handler is defined for every thread in the system. It is named *pager* after its main purpose of paging a thread. When receiving a page-fault IPC message, the pager selects a memory page and sends a reply with an embedded memory mapping. Upon receiving the reply the thread will continue executing. The page-fault resolution process happens transparently for the faulting thread. The mechanism for other exceptions works similarly. This mechanism allows for implementing arbitrary paging strategies in user-mode components without specific support in the microkernel itself.

4.2.4 Communication Channels

A communication channel is a special object provided by the kernel that allows two protection domains to use IPC to exchange data. Such a channel has a sender and a receiver side. A receiver binds to such a channel to be targeted as the recipient of a message by the kernel. The IPC sender cannot be directly identified by the receiver as in a system with local naming the receiver has no means to identify the IPC partner. Instead a label value identifies the channel through which the message is received. The label must be defined by the receiver upon binding to the channel, cannot be influenced by the sender and is visible to the receiver only. Multiple channels can be created when communication partners need to be distinguished.

For sending an IPC message, the sender invokes a capability to exchange data with the corresponding object. In case of the communication channel another user-mode implemented object is invoked and receives the message.

Neither the sender nor the receiver designate threads for communication but use communication channels instead. This enables to transparently interpose a communication relationship. For example, a server providing a basic memory allocator to clients might be interposed with a more sophisticated allocator that implements resource quotas. Neither the basic memory allocator nor the client will notice a difference except in timing behavior.

4.3 Runtime Environment

The runtime environment implements major operating system infrastructure and forms the first layer of user-mode services. Its goal is to abstract from the pure kernel provided functionality and offer generic and known interfaces to applications as well as implementing policies for those. The environment consists of separate components, called servers, and libraries to be used by applications. The runtime environment is called L4Re[4].

The root of all services is the roottask, which handles core resources such as memory. The roottask starts a single application, which is usually the application launcher. It is responsible for starting the remaining parts of the system.

The platform is configured and managed by an I/O component. It initially scans the platform for devices and then provides clients access to those devices. Access rights are exclusively granted by the I/O component based on a configuration that defines which devices or device classes can be accessed by which clients.

Functionality for accessing the servers is implemented in the provided libraries that contain the needed communication code. Standard functionality such as a subset of POSIX, as well as C, C++ and pthread libraries complete the set.

[4]http://os.inf.tu-dresden.de/L4Re/

4.4 Bootstrapping the System

In the following we will shortly describe the boot sequence on a typical device. Whenever the device is switched on, it will start at a defined location in the device and load a platform bootloader. This bootloader includes functionality to access internal storage, typically flash memory, where the OS image is stored. It will read this image into the device memory, verify its signature and finally launch it.

The OS image starts with the boot code that setups the initial components in memory and finally continues execution with the kernel. The kernel initializes itself and when done, starts the roottask. The roottask initializes itself and finally starts the launcher application, which boots the rest of the system. The launcher application is the first component that has the ability to load further applications from storage devices by using other services that provide device driver functionality.

To facilitate a secure boot chain, the initial platform loader needs to verify the OS image for authenticity. Given that, the launcher application needs to verify any application it loads from external media. The infrastructure for the cryptographic operations can be implemented in a TPM-like subsystem in the hardware, being a real TPM or a software implementation. In any case the cryptographic keys must be kept securely within the device.

4.5 Virtual Machines as Legacy Containers

Creating an OS from scratch is an uphill battle against existing OSes. Years of development and innovation went into the design and implementation of existing mobile OSes and their applications. Furthermore, they incorporate new features at a fast rate. Thus we want to leverage existing applications and their features within our framework.

However, applying a new OS construction mechanism to existing OSes is not feasible. Furthermore, due to resource constraints it is not feasible to port existing applications to a new OS. VMs encapsulate existing OSes together with their applications. This enables existing software and features on the new system. However, this setup does not improve on the security of the existing OS. Security sensitive applications are not implemented within the VM, but run as isolated components outside of it. Even if the VM gets compromised, it cannot tamper with the secure applications.

Our framework offers two different types of VMs. One is using rehosting and is applicable to nonvirtualizable platforms. The other leverages the hardware virtualization features of current x86 based mainstream CPUs. For our prototype we settled with virtualizing the Android OS because it is currently the most popular smartphone OS. However, our approach is applicable to other smartphone OSes as well[5]. Our Android VM is based on technology that virtualizes Linux. Therefore we first present these base technologies, and will then proceed to introduce our virtualized Android.

4.5.1 L[4]Linux

L[4]Linux is a port of the Linux kernel to our microkernel. It is binary compatible with Linux, which allows us to run ordinary Linux applications. It can be given access to a restricted set of hardware by the I/O manager but it can also run without any access to peripheral hardware. Besides

[5]For our virtualization technology, we need to modify the smartphone OS kernel. Due to restrictive licenses modifying some smartphone OSes is not allowed.

being used to run Linux applications, L^{4}Linux can also act as a service provider for other tasks outside of the virtual machine. An important use case for L^{4}Linux is to use it as a driver provider. For this, L^{4}Linux is given access to one specific device, and enhanced with an abstracted interface that enables data exchange with other L4Re tasks. This enables us to use devices without requiring the cumbersome work of reimplementing device drivers.

L^{4}Linux currently runs on the ARM and x86 architectures, and has shown good performance for many workloads [25]. It is available from its public website[6].

4.5.2 Karma VMM

Karma[7] is a VMM that runs as a task on top of the microkernel. It coordinates and controls a VM using the mechanisms provided by the microkernel (see Section 4.2). Karma strives to be of minimal complexity and thus resorts to aggressive paravirtualization. However, the changes required to a guest OS kernel are small compared to the changes required for L^{4}Linux.

To set up a VM, Karma creates a VM object. A portion of Karma's memory is mapped to the VM object as guest physical memory. Karma donates its processing time to the VM object by using a special system call to issue the world-switch. Karma regains control upon external events such as timer interrupts or the guest initiating a hypercall.

Each Karma instance hosts exactly one VM. Because of the isolation properties of the microkernel, a possible compromise of the VMM does not render other VMs compromised.

Karma currently supports a paravirtualized Linux as a guest OS. The changes to Linux are small and about 3.000 source lines of code (SLOC). The VMM requires a CPU with hardware virtualization support (SVM, VT-x) and has shown to offer exceptional performance [43].

As described in Section 2.2 hardware virtualization support has not found its way into the embedded market yet. However, we are confident that such features will soon be deployed in future smartphones and other high-end mobile devices. Intel announced the Moorestown successor Medfield to have VT-x support. ARM introduced the Cortex-A15 design [41]. This renders Karma an efficient alternative to L^{4}Linux on future devices.

5. L^{4}Android

The L^{4}Android project builds on the runtime environment and microkernel projects. L^{4}Android's goal is to run Android in a virtual machine on top of the microkernel. We offer two solutions for this, one being based on L^{4}Linux, the other based on the Karma VMM.

L^{4}Android is derived from L^{4}Linux and was created by adding code to L^{4}Linux that is required by Android. This code is provided by Google and comprises the binder IPC mechanism, wakelock support and the low memory killer.

The Karma-based version of L^{4}Android was created in a similar fashion. Here we applied the Karma Linux patch to the stock Android kernel provided by Google.

Both versions of L^{4}Android run the Android user-mode software stack as is, without further modifications. We tested the Android versions Eclair (2.1), Froyo (2.2) and Gingerbread (2.3).

L^{4}Android is an open source project, and its code repositories are accessible at its public website[8].

5.1 Prototype Implementation

We have implemented the core components of our framework for both the x86 and ARM architectures. Our prototype works on an ARM based iMX.51 board and an x86-based smartphone.

The ARM based iMX.51 board from Freescale features a 800MHz Cortex-A8 core that can be found in many cheaper smartphones today. The board is equipped with 512MB of RAM. On this platform we can run the microkernel, its accompanying runtime environment and a fully functional L^{4}Android.

Our framework also supports a developer phone from Aava Mobile. It is built around Intel's x86-based mobile CPU Moorestown. The CPU is clocked with up to 1.6GHz and has access to 512MB of RAM. Please refer to Figure 5 for a picture of this smartphone. It runs L^{4}Android side-by-side with L^{4}Linux.

6. EVALUATION

To show the generality of our framework, we will now describe how it can solve four important security scenarios. We show how our framework can be used to implement *Secure Software Smartcards* to be used for NFC applications or software SIM cards. The *Unified Corporate and Private Phone* scenario shows how the business and the private phone can be securely combined on one device. The third scenario describes how our framework can be used for *Mobile Rootkit Detection*, a technology that was previously restricted to desktop computers. The last example shows how the framework is used as an *Hardware Abstraction Layer* (HAL) to help device manufacturers in avoiding the GPL and to ease the update problem of Android.

6.1 Software Smartcard

Smartcard: Plastic card containing an integrated circuit that implements a general purpose processor and memory. Increasingly, smartcards also contain a number of coprocessors that accelerate special computations like cryptographic operations. Smartcards are used as debit cards, in access control systems, and SIM cards.

Recent smartphones contain NFC hardware. NFC has many potential uses, such as mobile payment, and ticketing [38]. Applications such as mobile payment require a trustworthy security anchor for their operation. Usually smartcards are used as a security anchor, as they provide a physically isolated execution environment together with a portion of memory that holds cryptographic key material. The cryptographic keys never leave the smartcard and therefore cannot be leaked by a compromised smartphone OS. The SIM card is a special smartcard. It is used to authenticate a subscriber to the mobile network. As SIM cards are available in all mobile phones, a natural decision is to use the SIM card as the security anchor for NFC as well. However, the SIM card is issued and controlled by the network operator. Many stakeholders, for example, financial institutes that want to make use of NFC, cannot give away

[6] http://l4linux.org/

[7] http://karma-vmm.org/

[8] http://l4android.org

control of key material to third parties. They have their own certification processes, and therefore rely on their own smartcards. Unfortunately, the number of installable (physical) smartcards is limited by the available space inside the smartphone's case.

A software-based implementation of smartcards on the application processor has the following merits:

Flexibility Creation and destruction of smartcards can be done instantaneously and without hardware modification. The software smartcard could be installed via download, which would be much more convenient than having the network operator send them via postal mail.

Multiple Instances Today's smartphone CPUs are powerful enough to house more virtual smartcards than are possible to install in hardware.

Consolidation The phone does not need to house many physical smartcards and their corresponding smartcard readers. This helps reducing the bill of materials and potentially make smartphones smaller and thinner.

A smartcard implemented on the application processor needs to be run in a secure environment. The TrustZone [44] technology implements an additional processor mode that establishes an isolated and secure environment. The TrustZone secure environment is populated via secure boot at system boot time, and cannot be exchanged at runtime. Usually smartphone vendors do not allow custom software in the TrustZone environment. Therefore TrustZone is not an option for secure software-implemented smartcards.

Instead, the software smartcard needs to be isolated in a secure environment that is established by the OS. The OS has to ensure that tasks running on the application processor cannot infer any information about the key material nor the cryptographic routines of a virtual smartcard. An authority that is to certify the virtual smartcard needs to make sure there are no implementation errors, which could be exploited to gain access to the smartcard. Correctness can only be asserted by software verification. Klein et al. [4] managed to formally verify a microkernel. Software verification hardly scales up to more than 10 thousand SLOC, which renders verification of a monolithic OS such as Linux infeasible.

We propose a system that is based on our framework. The microkernel consists of about 20.000 SLOC and can be certified. Furthermore the microkernel implements secure scheduling to mitigate timing-based side-channel attacks. In our framework Android cannot tamper with the software smartcards, and therefore does not need to be certified. Our solution is applicable to all current smartphones and does not require any hardware extensions.

A possible setup with three smartcards is shown in Figure 2. It consists of an Android partition that runs isolated from the three virtual smartcards. One virtual smartcard acts as a SIM card and does away with the need for a physical one. The other two virtual smartcards are used for NFC. The virtual smartcard multiplexer forwards requests between the NFC hardware and different virtual smartcards.

The TCB of a virtual smartcard is small. It consists of the microkernel, the memory manager, the IO manager, and the smartcard multiplexer.

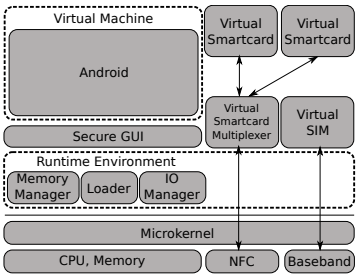

Figure 2: Setup running three virtual smartcards. They act as a softSIM and for NFC applications.

6.2 Unified Corporate and Private Phone

Smartphones are heavily used in today's businesses. Business executives use them to connect to their corporate IT. The phone is used to aggregate information critical to the business such as email, documents, and other PIM data. Unsolicited revelation of that data might cause business deals to fail or cause the companies stocks to crash.

Therefore business smartphones are usually provisioned by the corporation's IT departments. They make sure that the devices are configured according to the company's security policies. Additionally, the IT department installs certificates and cryptographic keys to make the phone connect to corporate VPNs. The devices might further be locked down to run only certified applications. Private use is often disallowed by the company's security policies.

Because of these measures, corporate smartphones are often cumbersome and inconvenient to use. Consequently many business executives carry an additional private phone with them. This is unnecessarily inconvenient and running both tasks on one device is desirable.

Using our framework, we can build a smartphone architecture that unifies the corporate and private phone in a secure way on one device. The setup is depicted in Figure 3. Two instances of L^4Android form two Android partitions where one is for private and the other for business purposes. The private partition is left as is. In fact we allow rooting of this partition, as it has no consequences for the corporate partition. The corporate partition can be hardened according to the business' security policies e.g. with SELinux [47].

Isolation has to be enforced not only between the two Android instances but at the user interface as well. Access to the graphics hardware and input devices such as touchscreen and buttons is multiplexed by a secure GUI server [37].

Access to devices shared by both Android instances needs to be multiplexed. We do this by virtualizing the hardware with a special device driver that implements multiple personalities, and presents the VMs with their own virtual devices. Virtualizing the baseband presents a particular challenge and is part of our current research.

For our demonstrator we decided to avoid the problem by assigning exclusive baseband access to one partition. This partition implements telephony and SMS functionality exclusively. The data connection is shared over a virtual Ethernet connection with the other Android partition.

Figure 5 shows our demonstrator running on the Aava developer phone. It runs an L^4Android partition side-by-side with an L^4Linux partition.

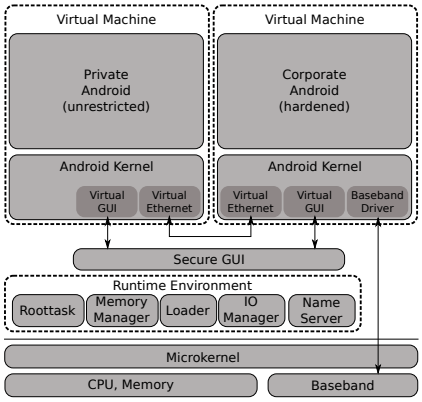

Figure 3: Setup implementing two isolated partitions on a mobile device. A secure hardened partition for business critical information and an open private one.

6.3 Mobile Rootkit Detection

Rootkits are a class of malware that is known to hide inside the kernel. A rootkit modifies the kernel's behaviour to hide itself, to open new backdoors, and to carry out other malicious activities. Recent work [26, 19, 12] showed that rootkits on smartphones are feasible and pose a serious threat to the users privacy and security.

Rootkit detection requires access to kernel memory. A compromised kernel must not mitigate rootkit detection. Therefore, rootkit detection software needs to be isolated from the kernel. Recent work presented rootkit detectors that either use special devices with DMA memory access [5], special coprocessors [48] or that reside inside a hypervisor [32]. These solutions operate in a layer below the target OS and therefore do not rely on the target OS kernel's integrity. Applying such approaches on a smartphone seems desirable, but was not feasible because there was neither special hardware nor virtualization in commodity smartphones.

With L^4Android, we provide a virtual machine to run Android. Therefore we investigate how existing hypervisor-based rootkit detectors are applicable to our system. The L^4Android architecture differs significantly from traditional hypervisor designs. Thus, further work is needed to port existing rootkit detection software.

6.4 Hardware Abstraction Layer

Android smartphones are sold by numerous manufacturers. The device models differ in their hardware. Depending on their target price, they come with different SoCs with different CPUs, clockings and total amounts of memory. Other distinctive components are the included screens, cameras and other peripherals.

The device manufacturers need to provide custom drivers for each of these components. The manufacturers strive to keep the innards of their devices a trade secret. However, with Linux being licensed under the terms of the GPL, they are obliged to provide drivers that reside in the kernel in source form. To avoid revealing to much intellectual property, many device drivers are split into two parts. One is an in-kernel proxy that is distributed in source form. It does not contain device logic, and is usually not contributed to Linux mainline. The other is a user-mode component that is distributed in binary form and contains all the device logic.

When Google issues an update containing a new kernel version, the device manufacturers need to port their drivers to the updated kernel's interfaces. Subsequently they have to go through the quality assurance processes. Both steps are time consuming, and the costs are often a prohibiting factor.

We propose to implement a generic hardware abstraction layer. The abstraction layer runs below the Android kernel and contains all device specific drivers. This setup is depicted in Figure 4. The Android kernel would contain only generic drivers that interface with the hardware abstraction layer. These generic drivers are the same on all smartphones. Thus, a new kernel version can be provided by Google, and can be deployed on all Android smartphones without further involvement of the device manufacturer. This allows for much faster deployment of security critical kernel updates.

Having the drivers below Android solves the problem with the GPL license of the Linux kernel. As they are implemented below Android, they do not need to be released in source form. Therefore the vendors do not need to resort to split drivers, which reduces their complexity. The generic drivers in the Android kernel can be contributed to Linux mainline.

Device drivers in the hardware abstraction layer are implemented as individual tasks, each running in its own address space. A potential driver crash is confined within this address space. Using watchdogs, the system can detect a driver crash, and restart the driver. Such a crash would even be transparent to Android.

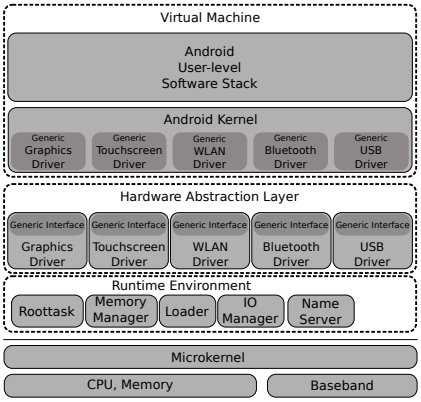

Figure 4: Proposed architecture containing a generic hardware abstraction layer. Device specific drivers are implemented in its own layer. The Android kernel implements a generic driver interface that is the same on all devices.

7. RELATED WORK

Related work comprises microkernel research, mobile virtualization and efforts to improve Android security.

OK Labs implemented the microkernel OKL4 [3], which is used in the Motorola Evoke. It is used to consolidate application and baseband processors. In contrast to our solution OKL4 is proprietary, and closed source.

Klein et al. [4] designed and implemented seL4. SeL4 is the first microkernel formally verified to implement its specification. This microkernel is tuned for verification and does not implement multiprocessor support. Similar to OKL4, the seL4 kernel was not released in source form.

In contrast to L4Re, which was specifically developed as the user-mode counterpart to the Fiasco.OC kernel, Genode [2] is a framework designed to support a variety of kernel interfaces. Fiasco.OC is supported as one of 8 different kernels. Even though Genode supports the use of paravirtualized Linux, its primary vision is a small component-based general purpose OS rather than a virtualization platform. It strives to replace UNIX-based OSes in application domains where high security requirements meet dynamic workloads.

Schmidt et al. [8] show how a mobile trusted module (MTM) can be implemented on a microkernel based system. The same authors also show how virtual SIMs can be implemented in such a system [35]. This work covers the SIM only.

VMware introduced a mobile virtualization solution. It is designed to run an Android VM on top of Android. The setup implements a Type-2 hypervisor [9] and the integrity of the VM relies on the integrity of the host OS kernel. VMware's emphasis is on manageability and it does not improve on security. Hwang et al. [29] reported on a port of Xen to ARM platforms. In Xen all VMs depend not only on the hypervisor, but on a full OS and its applications that act as Dom0. With a full OS in their TCB, VMs on top of Xen are not an option for highly secure applications.

Enck et al. [46] implemented an online information flow tracking system on Android. Beresford et al. [7] mocked hardware resources to revoke application's access to particular resources at run-time. Zhang et al. [47] applied mandatory access control to Android with SELinux. Zhou et al. [49] implemented a custom privacy mode to enable fine-grained control over application's access on private information. All these solutions improve on Android security, but fall short if the kernel is compromised.

8. CONCLUSION AND FUTURE WORK

In this work we presented a generic OS framework that facilitates the creation of secure smartphone systems. The framework consists of three core components. A microkernel acts as the secure foundation and is accompanied by a user-mode runtime environment. The third component are VMs to securely encapsulate existing smartphone OSes.

We implemented the core components of our framework on a mobile x86 and ARM platform. This implementation is available as the open source L^{4}Android project. We evaluated our framework by showing how it can be applied to solve four challenges in smartphone security such as secure software smartcards, and a unified corporate and private mobile phone. Our framework does not rely on special hardware features and is thus readily applicable to a wide range of existing smartphones.

The security demands of future smartphone applications cannot be fulfilled by existing smartphone OSes. Fixing the security problems of those OSes is infeasible. Instead we demonstrated that our framework succeeds in creating a secure execution environment for applications with high demands on security. Thereby it enables future use-cases of smartphones, such as NFC based mobile payment.

8.1 Future Work

Future work comprises three areas: power efficiency, hardware accelerated graphics, and design and implementation of a modern secure GUI for smartphones.

Power Efficiency As a first step we have to assess the power efficiency of our system, in setups that differ in the number of virtual machines and in the number of secure tasks. The second step is then to improve upon these numbers, and apply proper power management facilities.

Hardware Accelerated Graphics Today our framework comprises a secure GUI that relies on software rendering. Software rendering consumes precious computing time from the CPU and leaves the powerful graphics processors idle. To decrease CPU utilization and improve overall responsiveness, we are working on a secure hardware accelerated GUI.

Support more Smartphone OSes We work on the support of more smartphone OSes. As a study, we ported the OpenBSD[9] kernel on our microkernel [10]. We are investigating if the iOS kernel allows for a port as well.

Apart from improving the framework, we are looking forward to applying it to manifold security problems. We are working on bringing honeypots to the world of smartphones [11].

Acknowledgements

We would like to thank Collin Mulliner for his instructive comments and support. We also thank Dmitry Nedospasov for his thorough review of the paper.

9. REFERENCES

[1] DroidDream. http://www.androidpolice.com/2011/03/01/the-mother-of-all-android-malware-has-arrived-stolen-apps-released-to-the-market-that-root-your-phone-steal-your-data-and-open-backdoor/, March 2011.

[2] Genode OS Framework. http://genode.org, June 2011.

[3] C. Walker and M. Konstant. OK Labs Enables World's First Virtualized Smartphone, with Mobile Virtualization Solution. http://www.ok-labs.com/releases/release/ok-labs-enables-worlds-first-virtualized-smartphone-with-mobile-virtualizat, 2009.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and H. Winwood. seL4: Formal Verification of an OS Kernel. In *ACM Symposium on Operating System Principles* (2009), ACM, pp. 207–220.

[5] A. Baliga, V. Ganapathy and L. Iftode. Detecting Kernel-level Rootkits using Data Structure Invariants. *IEEE Transactions on Dependable and Secure Computing 99*, Preprints (TBD 2010), TBD.

[9] http://l4openbsd.org

[6] A. Lackorzynski and A. Warg. Taming subsystems: capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (2009), IIES '09, ACM, pp. 25–30.

[7] A. R. Beresford, A. Rice, N. Skehin and R. Sohan. MockDroid: trading privacy for application functionality on smartphones. In *12th Workshop on Mobile Computing Systems and Applications* (March 2011).

[8] A.U. Schmidt, N. Kuntze and M. Kasper. On the deployment of mobile trusted modules. In *Wireless Communications and Networking Conference, 2008. WCNC 2008. IEEE* (31 2008-april 3 2008), pp. 3169 –3174.

[9] Barr, K., Bungale, P., Deasy, S., Gyuris, V., Hung, P., Newell, C., Tuch, H., and Zoppis, B. The vmware mobile virtualization platform: is that a hypervisor in your pocket? *SIGOPS Oper. Syst. Rev. 44* (December 2010), 124–135.

[10] C. Ludwig. Porting OpenBSD to Fiasco. `http://www.isti.tu-berlin.de/fileadmin/fg214/finished_theses/cludwig/OpenBSDonFiasco.pdf`, June 2011. Bachelor's Thesis, Security in Telecommunications, Technische Universität Berlin.

[11] C. Mulliner, S. Liebergeld and M. Lange. HoneyDroid - Creating a Smartphone Honeypot. In *Poster session of the IEEE Symposium on Security and Privacy* (May 2011), IEEE.

[12] C. Papathanasiou and N. J. Percoco. This is not the droid you're looking for... `http://www.defcon.org/images/defcon-18/dc-18-presentations/Trustwave-Spiderlabs/DEFCON-18-Trustwave-Spiderlabs-Android-Rootkit-WP.pdf`, July 2010.

[13] Canalys. Android increases smart phone market leadership with 35 share. `http://canalys.com/pr/2011/r2011051.html`, 2011.

[14] Coverity Inc. Coverity Scan 2010 Open Source Integrity Report. `http://www.coverity.com/html/press/coverity-scan-2010-report-reveals-high-risk-software-flaws-in-android.html`, 2010.

[15] D. Barrera, H. G. Kayacik, P. C. van Oorschot and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 73–84.

[16] D. Maslennikov. Malware in the Android Market: here we go again. `http://www.securelist.com/en/blog/11267/Malware_in_the_Android_Market_here_we_go_again`, 2011.

[17] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung and T. Jaeger. Measuring integrity on mobile phone systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies* (New York, NY, USA, 2008), SACMAT '08, ACM, pp. 155–164.

[18] Droid Life: A Droid Community Blog. Motorola Eases Up on Locked Bootloader Stance, Plans to Unlock Portfolio in 2011? `http://www.droid-life.com/2011/04/26/motorola-eases-up-on-locked-bootloader-stance-plans-to-unlock-portfolio-in-2011/`, April 2011.

[19] E. Monti. Iphone Rootkit? There's an App for That. `http://sandiego.toorcon.org/index.php?option=com_content&task=view&id=48&Itemid=9`, October 2010.

[20] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 41–41.

[21] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17* (July 1974), 412–421.

[22] G. Portokalidis, P. Homburg, K. Anagnostakis and H. Bos. Paranoid Android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACSAC '10, ACM, pp. 347–356.

[23] Google Inc. Distribution of Android Versions. `http://developer.android.com/resources/dashboard/platform-versions.html`, May 2011.

[24] Google Inc. Wallet. `http://www.google.com/wallet`, June 2011.

[25] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter and S. Schönberg. The performance of µ-kernel-based systems. In *Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 66–77.

[26] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy and L. Iftode. Rootkits on smart phones: attacks, implications and opportunities. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications* (New York, NY, USA, 2010), HotMobile '10, ACM, pp. 49–54.

[27] J. Liedtke. On micro-kernel construction. In *Proceedings of the fifteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 237–250.

[28] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9* (Berkeley, CA, USA, 2000), USENIX Association, pp. 10–10.

[29] J. Y. Hwang, S. B. Suh, S. K. Heo, C. J. Park, J. M. Ryu, S. Y. Park and C. R. Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE* (2008), IEEE, pp. 257–261.

[30] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ASPLOS-XII, ACM, pp. 2–13.

[31] K. Mahaffey. Security Alert: DroidDream Malware Found in Official Android Market. `http://blog.myloo`

kout.com/2011/03/security-alert-malware-found-in-official-android-market-droiddream/, 2011.

[32] L. Litty, H. A. Lagar-Cavilla and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th conference on Security symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 243–258.

[33] L. Xie, X. Zhang, J.-P. Seifert and S. Zhu. pBMDS: a behavior-based malware detection system for cellphone devices. In *Proceedings of the third ACM conference on Wireless network security* (New York, NY, USA, 2010), WiSec '10, ACM, pp. 37–48.

[34] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[35] M. Kasper, N. Kuntze and A.U. Schmidt. Subscriber authentication in cellular networks with trusted virtual sims. In *Advanced Communication Technology, 2008. ICACT 2008. 10th International Conference on* (feb. 2008), vol. 2, pp. 903 –908.

[36] Mulliner, C., Vigna, G., Dagon, D., and Lee, W. Using Labeling to Prevent Cross-Service Attacks Against Smart Phones. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (Berlin, Germany, July 2006), vol. 4064 of *LNCS*, Springer, pp. 91–108.

[37] N. Feske and C. Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference* (2005), IEEE Computer Society, pp. 85–94.

[38] O. Falke, E. Rukzio, U. Dietz, P. Holleis and A. Schmidt. Mobile services for near field communication. *Ludwig-Maximilians-Universität (LMU), Munich, Germany, Technical Report LMUMI-2007-1* (2007).

[39] P. Chou. HTC to open bootloaders. `https://www.facebook.com/HTC/posts/10150307320018084`, May 2011.

[40] R. Bhardwaj, P. Reames, R. Greenspan, V. S. Nori and E. Ucan. A Choices Hypervisor on the ARM architecture. *Department of Computer Science, University of Illinois at Urbana-Champaign 11* (2006).

[41] R. Mijat and A. Nightingale. Virtualization is Coming to a Platform Near You. *ARM White Paper* (2011).

[42] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung and L. Smith. Intel Virtualization Technology. *Computer 38* (May 2005), 48–56.

[43] S. Liebergeld. Lightweight Virtualization on Microkernel-based Systems. `http://os.inf.tu-dresden.de/papers_ps/liebergeld-diplom.pdf`, January 2010. Diploma Thesis, Chair of Operating Systems, Technische Universität Dresden.

[44] T. Alves and D. Felton. TrustZone: Integrated hardware and software security. Tech. rep., ARM Limited, 2004.

[45] W. Enck, D. Octeau, P. Mcdaniel and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium* (2011).

[46] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[47] X. Zhang, J.-P. Seifert and O. Acicmez. SEIP: Simple and Efficient Integrity Protection for Open Mobile Platforms. In *Information and Communications Security* (2010), vol. 6476 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 107–125.

[48] X. Zhang, L. van Doorn, T. Jaeger, R. Perez and R. Sailer. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 2002), EW 10, ACM, pp. 239–242.

[49] Y. Zhou, X. Zhang, X. Jiang and V. W. Freeh. Software Creates Privacy Mode To Help Secure Android Smartphones. `http://news.ncsu.edu/releases/wms-jiang-tissa/`, 2011.

APPENDIX

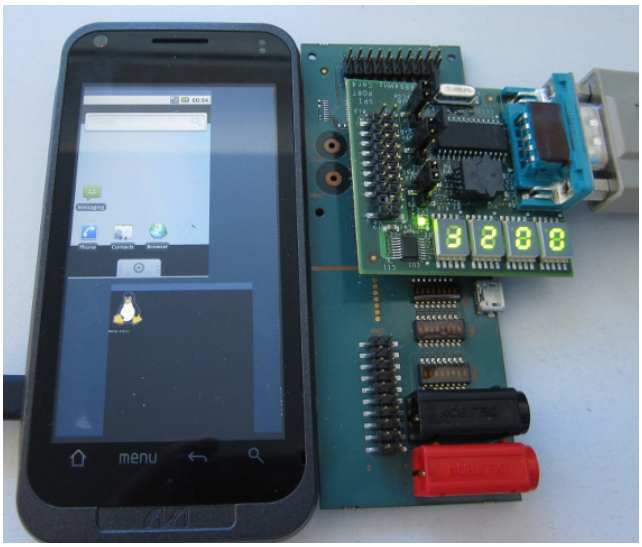

Figure 5: This is one of our Aava devices. It runs a VM with Android and another VM hosts standard Linux. A debug board is attached to the right, which provides a serial line.