Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Johannes Wilken

Dynamic context-based execution control of Applications on mobile devices

Fakultï¿½t Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Johannes Wilken

Dynamic context-based execution control of Applications on mobile devices

Bachelorarbeit eingereicht im Rahmen der Bachelorpruefung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultï¿½t Technik und Informatik
der Hochschule fi¿½r Angewandte Wissenschaften Hamburg

Betreuender Prï¿½fer: Prof. Dr. Westhoff
Zweitgutachter: Prof. Dr. Martin Huebner

Eingereicht am: 15. August 2012

Johannes Wilken

Thema der Arbeit
Dynamic context-based execution control of Applications on mobile devices

Stichworte
Smartphone, Android, Apps, Watchdog, MTM, Contextbasiert, Dynamisch, AusfÃ¼hrungskontrolle

Kurzzusammenfassung
Diese Arbeit zeigt eine Möglichkeit die Ausführung von Programmen auf mobilen Endgeräten auf Basis eines vordefinierten Kontexts zu kontrollieren. Hierbei kommt ein MTM zum Einsatz, welches die Basis der Architekture bildet.
Johannes Wilken

Title of the paper
Dynamic context-based execution control of Applications on mobile devices

Keywords
Smartphone, Android, Apps, Watchdog, MTM, context-based, dynamic, execution-control

Abstract
This thesis shows a possibility to control the execution of application on mobile devices based on predefined contexts. A MTM is used as it provides the base of the architecture.

Contents

1 Introduction

Nowadays, mobile devices are widely used in many different situations. However, not in all these situations it's welcomed to use such a mobile device for some activities.

1.1 Thesis Characterisation

This thesis describes an approach on how to control the execution of certain applications based of a context a mobile device is in. A context may be defined by anything suitable, such as a wireless network or by GPS[1], for locations, or a timeframe, e.g. from 8 am to 4 pm. When the device enters a prior defined context, it prevents the execution and terminates running applications which are not allowed in that context.

The *first chapter* describes the motivation of this thesis, as well as the preliminary work.

The *second chapter* introduces the Mobile Trusted Module (MTM)

The *third chapter* covers the security architecture.

The *fourth chapter* shows the Android OS.

The *fifth chapter* describes the implementation of the security architecture.

The *sixth chapter* describes the implementation of the context recognition.

The *seventh chapter* provides a security analysis.

The *eighth chapter* evaluates the architecture and gives an outlook to future work.

The work at the security architecture, the implementation and management application are attributed to the BMBF Project SKIMS.

1.2 Motivation

Mobile devices, like smartphones, are able to serve a lot of purposes, not limited to making phone calls and mobile internet. They can be used as assistant in many ways, like guiding people to their destination or offering additional information via augmented reality. As an

[1] (Global Positioning System), a space-based satellite navigation that provides location and time information (Wikipedia, 2012h)

example, a company, or similar institute, could use a smartphone to stay in touch with their employees and distribute work among or communicate with them in a very simple way.

However, due to their many capabilities, an employee might be distracted by his smartphone and instead of assist him during work it impedes him. To avoid such a situation, a company may want to disable certain applications while the employee is somewhere on the company area. This would allow to give smartphones as work assistant to them without being concerned about them getting distracted.

Another important use case is preventing an employee to use a mobile device for taking pictures or creating copies of secret, company intern documents. This could be done by controlling the execution of a camera application during the time of work. Some employers may want to prevent their employees to communicate with each other using a mobile device during a certain time, like an examination. In that case, an employer could set up a context for the examination and prevent the communication by disabling services like SMS or calling.

1.3 Goal

The proposed solution provides the ability to disable launching applications not welcomed in a particular location. Furthermore, if such an application is already running when entering, this application will be terminated gracefully. This is to prevent data loss due to pending work, e.g. I/O-Operations. A disabled application should never run and the solution should not be possible to be bypassed in any way.

2 Mobile Trusted Module

The hardware base for this thesis is a MTM[1], a Mobile Trusted Module. This module was specified by the Trusted Computing Group (TCG)[2] and its main concept will be covered in this chapter. The specification describes the MTM as an aggregation of functions and commands to ensure trustworthiness. It was specified to provide a comparable solution for this problem as does the Trusted Platform Module (TPM)[3] for laptops and PCs. In detail, a MTM consists of two separate parts, the MRTM[4] and MLTM[5], which will be covered in the following sections. This module is capable of taking and storing metrics from hardware as well software, such as hash values, and compare them with reference values. These values are stored inside the MTM and, due to complete separation from any other hardware of the device, can't be altered or manipulated in any way. Hence, it is possible to securely use these values to ensure the integrity of the device. In order to protect the MTM itself from being manipulated, which would possibly break the security granted by it, it is able to measure and confirm its own integrity. To secure the measurements for discrete integrity, a Platform Configuration Register (PCR) is used. A PCR is a shielded location inside the MTM to which only the module has access. The specification of the MTM contains to core domains, which are fitted for different stakeholders. The difference between these two lies in utilisation and accessibility to the device that is used. The first stakeholder is the local-owner, which is, as usual, the user of the device and thus has physical access, meaning he can execute and load any software he wishes. The other stakeholder is the remote-owner, which does not have physical access to the device, but provides services to it and needs to ensure that their engines are functioning properly. This is done through a secure boot process, which is elaborated later in this chapter. The configuration of the MTM is done during manufacturing with a set of configuration data matching the data

[1](Mobile Trusted Module), a module that provides security features to mobile devices, such as platform integrity and device authentication (Trusted Computing Group, 2010, p. 14)

[2](Trusted Computing Group), an international industry standard group that develops specification for trusted computing (Trusted Computing Group, 2012)

[3](Trusted Platform Module), a module to provide basic security features

[4](Mobile Remote-Owner Trusted Module), a MTM that provides a subset of the commands specified by the TPM version 1.2 and additionally mobile specific commands (Trusted Computing Group, 2010, p. 14)

[5](Mobile Local-Owner Trusted Module), a MTM that provides a subset of the commands specified by the TPM version 1.2 and optionally additional commands (Trusted Computing Group, 2010, p. 14)

from the stakeholders' scope, either inside the Mobile Local-Owner Trusted Module (MLTM) or inside the Mobile Remote-Owner Trusted Module (MRTM). (Trusted Computing Group, 2010, p. 13)

2.1 Mobile Local-Owner Trusted Module

This module covers the local-owner, usually the user of the device. It has to support a subset of the commands of a TPM version 1.2 and thus provides comparable features to the mobile device as the TPM provides to PCs. The MLTM is designed to support remote verification, like remote attestation. This can be used by a software company which provides software for that security is crucial, e.g. online banking software, to detect if the user of the device has tampered with the software in any way to corrupt the security. In addition, the MLTM can also be used to provide local verification as a request of a local owner. In this case, the local owner can also be a software that is used to take the measurements of other software that is currently running and compares them to stored measurements. (Trusted Computing Group, 2010, p. 13-15, 38, 62)

2.2 Mobile Remote-Owner Trusted Module

Contrary to the MLTM this module is designed for a scenario in which the owner is a remote party and thus does not have physical access to the device. This means that physical presence authorisation must not be supported in the MRTM, however it must be supported in a MLTM. Another difference between a MRTM and a MLTM is that the latter does not have to provide mobile specific commands and the MRTM contains additional capabilities to support a secure boot process. However, just as the MLTM, a MRTM must support a subset of the TPM v1.2 commands. In order to support local verification, a MRTM is required to support commands to install RIMs[6] and to verify them. These commands are optional for a MLTM. This means, that a MRTM is able to generate internal RIM certificates from an external RIM certificate. (Trusted Computing Group, 2010, p. 13-15, 38)

2.3 Secure Boot

The MTM is used to store reference values for a secure boot procedure. This procedure ensures that the device is starting from power-on in a trustworthy state and later these reference values

[6] (Reference Integrity Metrics), is a reference value to compare a measurement against, for instance a SHA256 hash of a software image Trusted Computing Group (2010)

are also used to shut the device down in a trustworthy state. The secure boot establishs the chain of trust. If any of the checks during the secure boot fails, the boot process is terminated. The figure 2.1 illustrates the secure procedure, which consists of the following steps:

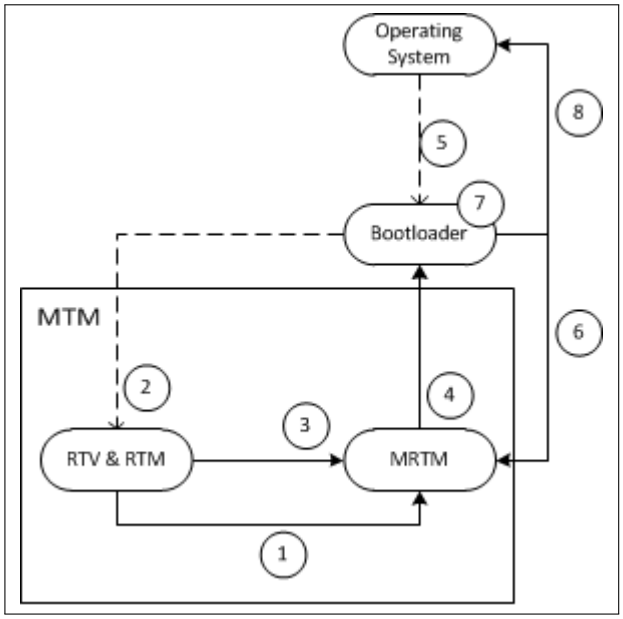

Figure 2.1: The MTM secure boot procedure (Trusted Computing Group, 2010, p. 15)

1. The MTM firstly measures its own integrity and verifies it by executing a piece of code stored inside the MTM.

2. The modules involved in this are the Root-of-Trust-for-Verification (RTV) and the Root-of-Trust-for-Measurement (RTM). The latter calculates metrics of itself and the RTV, which then verifies these metrics from the RTM. Lastly, the bootloader is measured is verified by these modules.

3. To verify the bootloader an actual metric from the bootloader is taken by these modules and is compared to a reference value. The reference value was also taken from the bootloader, when it was in a secure, trustworthy state, for example at manufacture time. When the actual metric and the stored reference value are matching, it verifies the bootloader to be in a secure and trustworthy state.

4. When it's ensured that the bootloader can be trusted, any further measuring task is delegated to it.

5. The bootloader then measures the integrity of the OS kernel.
6. These measurements are passed to the MTM for comparison.
7. The MTM compares the measurement it received from the bootloader with a reference measurement again taken in a trustworthy state. The result is then passed back to the bootloader.
8. When the kernel is verified by the MTM, the bootloader starts the kernel.

2.4 Reference Integrity Metric Certificate

To provide reference metrics and additional information for a entity, such as a binary of an application, a Reference Integrity Metric (RIM) is used. As shown in the last section, reference metrics are used to verify the integrity by measuring entities and comparing them to already taken reference metrics. These reference metrics are called RIMs. To protect a RIM against manipulation, it is stored inside a PCR. A PCR is a secured storage only accessible by the MTM. To ensure that the RIM origins from a trustworthy source, it is certified before it is stored inside a PCR. These RIM certificates can be a signed hash of the RIM itself plus additional information containing data of the origin and the trust of this certificate, like a shared secret. RIM certificates can be generated in two ways, inside the MTM and outside. As the MTM can be considered trustworthy, as it would fail the secured boot procedure check otherwise, certificates generated inside the MTM are automatically considered trustworthy. Additionally, RIMs generated inside a MTM are bound to this specific MTM and cannot be used in a different one. This restriction does not affect external generated RIM certificates, these can be used across multiple platforms, however, they can be converted to internal RIM certificates if needed. External RIM certificates are needed to be verified by internal RIM certificates or other authentification methods before they can be accepted. (Trusted Computing Group, 2010, p. 18-19)

2.4.1 Keys

RIM certificates are secured by several keys used by the MTM. The used keys are described here.

Verification Key

The verification key is a public key and can only be used by the MTM for verification purposes. A verification key can be represented in a hierarchical authorisation. The root key in this hierarchy is called *Root Verification Authority Identifier*. This key enables to verify other verification keys in the hierarchy.

Endorsement Key

This key is used to sign RIM certificates. It is the private key of a private-public-key-pair with the public key being the verification key. This key is unique to each MTM and is only known to authorised parties which are allowed to create RIM certificates. Depending on the mode of the MTM, whether it's in MRTM or MLTM configuration, this key is stored in a PCR, outside the MTM or both. (Trusted Computing Group, 2010), (Trusted Computing Group, 2007)

Storage Root Key

This key is sealed inside the MTM and bound to an owner. When the owner changes, a new SRK is generated for the new owner (Trusted Computing Group, 2007). It is used to sign certificates that provide new verification keys (Trusted Computing Group, 2010).

Attestation Identity Keys

This key is used by the MTM to sign measured PCR values. The signed values are used to verify an intact integrity of the transmitted values to a remote service. This service is then able to verify and trust the integrity of the values and treat them as a trusted measurement (Trusted Computing Group, 2010, p. 8).

2.4.2 Counters

Additional to the keys, the MTM is protected by counters. Both of these counters are placed in PCRs to protect them against manipulation and are only increased when all operations affecting the counter values were performed successfully.

counterRIMprotect

This counter protects the internal RIM-certificates against re-flashing. A RIM certificate provides a *"counter-stamp"* field that is compared to the actual counterRIMprotect counter value to ensure the freshness of the certificate. The counter is monotonic, it can only be

increased, and its proposed limit is 4095. Any RIM certificate having a lower value to the actual counterRIMprotect counter value become outdated and thus untrusted (Trusted Computing Group, 2010, p. 34).

CounterBootstrap

This counter protects the device against installing a new firmware. Like the counterRIMprotect counter, this counter is as well monotonix and outdates RIM certificates for the old firmware image. The new firmware is validated by an external RIM certificate that comes with it. The proposed limit for this counter is 31.

3 Security Architecture

The security is provided by the concept architecture introduced in this chapter. The concept is orthogonal to the system installed on the mobile device, however it has requirements and boundaries which are discussed as well. The architecture consists of five major parts creating a working architecture. These major parts are:

- Secure Startup
- Program Certificate Verification
- Program Execution Control
- Program Certification
- Context Recognition

This extends the security architecture proposed in (Landsmann, 2011) by the element of context recognition. Similar concepts are introduced in (Trusted Computing Group, 2010) and (Ugus und Westhoff, 2011).

3.1 Security Goal

The security architecture provides a secure startup, to start a mobile device in a secure and trustworthy state. This is done to prevent uncertified code from execution. Applications that have been altered in any way, even by an update, as these can change the behaviour of an application, can be detected by this architecture and prevented from running. In addition, this architecture creates certificates and handling of these directly on the device, which makes them uniquely usable with it (Landsmann, 2011). As this architecture is intended to secure the device from malicious software, only legitimated users, such as administrators, are allowed to request program certification. Furthermore, this certification should be impossible to bypass. In addition, the architecture should be hardware based and provide a secure key storage to secure the keys against manipulation or extraction. Also, to reduce any computational

overhead, it should aid the security mechanism of the present system. This lowers the power consumption of the mobile device compared to other solutions like anti-malware programs, which often use scanning to detect malicious software (Landsmann, 2011).

3.2 Secure Startup

The first stage of the architecture is a secure startup. This ensures that the mobile device boots in a well-known and trustworthy state and provides a trust anchor. This means a chain of trust is established directly at the start of the device. The first step is to check the hardware for any manipulations. Secondly, the flash image of the device has to be checked for manipulations as well as the bootloader for the OS on the device as the third step. These checks are performed by using metrics that were taken and calculated earlier in a secured and trustworthy state and comparing them to metrics the system takes at startup from the flashed image, bootloader and operating system. The trustworthy metrics are stored inside a secured, immutable storage, protecting them against manipulation and only accessible once the secure startup has finished and the device is in a trustworthy state. To further enhance the security, the metrics are encrypted, protecting them against recreation of equal entries to allow malicious application to run. This procedure requires a hardware module to be present that is capable of taking metrics, providing measuring and cryptography as well as a secure storage to save these values. This hardware would hence establish the chain of trust, independed from the system running on the mobile device, as it confirms that the mobile device hardware, as well as the software participating in the boot process (e.g. the bootloader and the OS itself) is trusted (Landsmann, 2011).

3.3 Program Certification

To determine which applications are allowed to run on a mobile device the architecture provides program certification. This is exclusively allowed to the administrator of the mobile device, so only a administrator is granted the control of which application are allowed to be executed. This requires the hardware to provide authentication and verification to ensure only these person can certificate programs for the mobile device. A program is certificate by providing data to represent the program, for example a hash of the binary, and other information about the program to the cryptographic hardware. The hardware then uses the provided information to created a certificate and signs it with its private key. This key, as well as the public key, is sealed inside the hardware module. However, the public key is revealed

only after the secure startup was successful to be used at runtime for program verification. (Landsmann, 2011)

3.4 Program Verification

A program is verified by computing its hash value. If the program was previously certified by an administrator, it's certificate will be found in the list of all available certificates on the device. As all the certificates are signed with the private key from the hardware module, the certificates can be stored inside unprotected storage. When altering these certificate, they would become invalid and the corresponding program won't be executed anymore. This requires the algorithm used to sign the certificates with the module's private key to be cryptographic strong so it is not possible to manipulate or recreate certificates successfully. Once the corresponding certificate is found on the device, the program is verified using its hash value and comparing it to the certificate, which is temporarily decrypted with the public key. (Landsmann, 2011)

3.5 Program Execution Control and Verification at Runtime

As the device is wanted to be kept in a trustworthy state, the chain of trust established with the secure startup needs to kept valid. This is done through verification at runtime. The OS handles every program execution and thus has to verify programs before executing them. Only if the verification was successful, the OS is allowed to execute the program, otherwise the executed will be aborted immediately. As the verification is unsuccessful when no corresponding certificate was found, meaning the program has either not been certified by an administrator or was altered since the certification, this prevents malicious programs from being executed on the device and thus protecting it (Ugus und Westhoff, 2011).

3.6 Context Recognition

The last part in this architecture covers the ability to make the security context sensitive. This means, that programs can be certified for specific contexts. Contexts can be bound to lists of certified programs which are allowed to be executed in the context. As with program certification, the context creation is only granted to the device administrator. A context can be anything the device administrator considers fitting, the architecture does not put any

restriction on this. This means, a context can be defined with a wireless network, a specific time, an incoming call, a check-in at a NFC station or a combination of multiple types.

4 Android OS

The Android OS[1] is designed for mobile devices. It's build upon a Linux[2] kernel[3], however, it has stricter energy policies due to battery limitations. As well as the Linux kernel, the Android kernel is open source software[4] and can be obtained and modified. This is the main reason why the Android OS is chosen for an example implementation since we need to modify the kernel to implement some of the needed features.

4.1 Android Architecture

The Architecture of the Android OS is parted in 5 layers, the Linux kernel itself as the bottom-most layer, on top of that is the library layer, next to the android runtime, the application framework layer and the application layer are the top-most layers in this architecture. The Linux kernel and the library layer are both implemented in a low-level programming language, like C programming language[5] orC++ programming language[6], whereas the other 3 layers are implemented in Java programming language[7].

4.1.1 Linux Kernel

Core services, like security, memory and process management, network stack and driver model, are provided by the Linux kernel version 2.6. This also serves as an abstraction layer between the actual hardware of the device and the rest of the software stack.

[1] (operating system), a set of software that manages computer hardware and provides services for other software (Wikipedia, 2012q)
[2] a modular, open source software OS (Wikipedia, 2012o)
[3] the main component of a OS, a bridge between applications and actual data processing at hardware level (Wikipedia, 2012n)
[4] computer software that is available in source code form (Wikipedia, 2012p)
[5] a low level, general-purpose programming language (Wikipedia, 2012e)
[6] a low level, general purpose and object oriented programming language (Wikipedia, 2012d)
[7] an object oriented programming language(Wikipedia, 2012m)

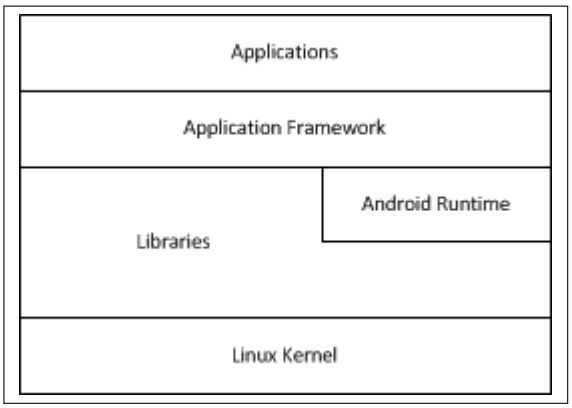

Figure 4.1: Android OS Architecture (Google Inc., 2008a)

4.1.2 Libraries

This layer provides essential libraries to the software stack, such as various media libraries or implementations to render graphics or fonts. These libraries are implemented in the C programming language or the C++ programming language and are used by various components of the Android system.

4.1.3 Android Runtime

The Android runtime executes the various application that may be installed on a devices. It contains the DVM[8] and a set of core libraries which provides most of the functions available in the core libraries in the Java programming language. An application will always run in its own Process[9] and within its own DVM.

4.1.4 Application Framework

This layer provides the libraries from the library layer as various Java frameworks to the user.

4.1.5 Application

The top-most layer includes the applications installed on the device. All these applications are written in the Java programming language and are executed in the DVM.

[8] (Dalvik Virtual Machine), a stack-based virtual machine for Java programming language used in the Android OS (Wikipedia, 2012f)

[9] an abstract instance of a computer program that is currently running (Wikipedia, 2012r)

4.2 Boot Sequence

Since the Android OS is based on the Linux kernel, the boot sequence is similar to that of the standard Linux kernel. At first, a bootloader will initialise very low-level systems before loading the actual Linux kernel. This kernel then initialises the hardware, driver and files.

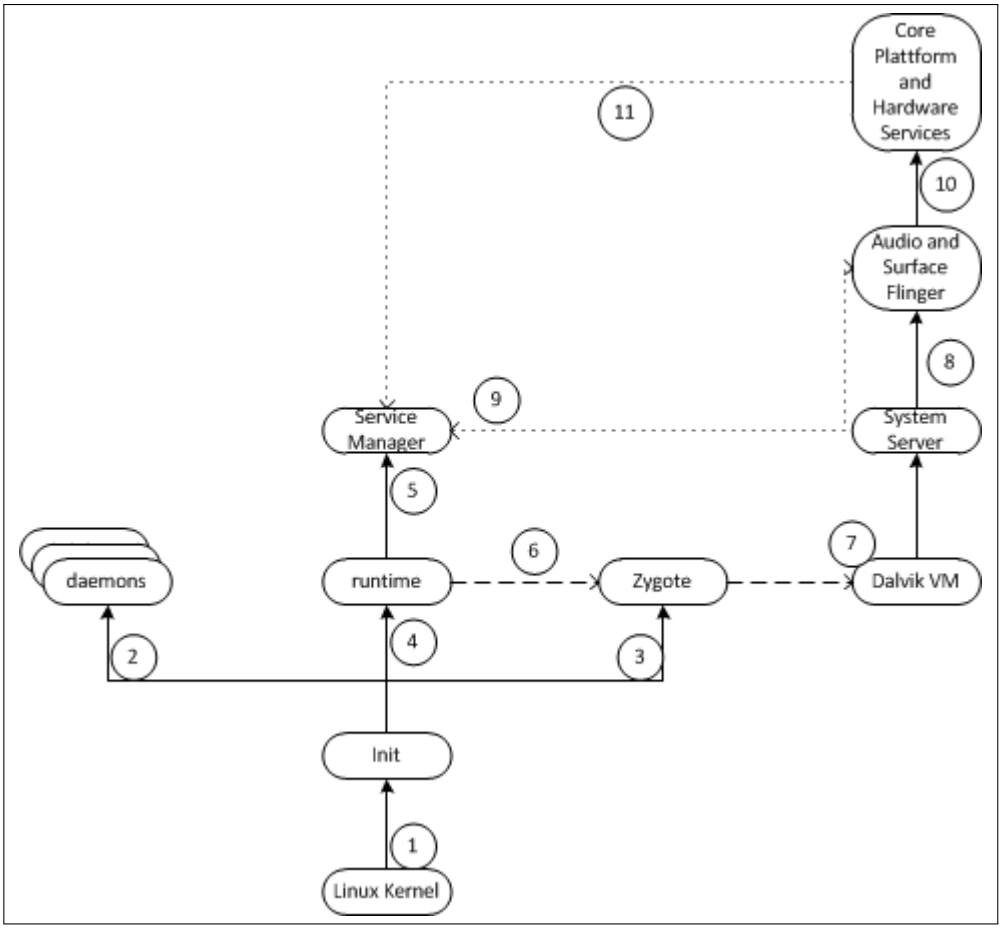

Figure 4.2: Android OS Boot Sequence (Google Inc., 2008a)

1. At first, the bootloader loads the Linux kernel and starts the Init process.
2. The Init process then starts Linux daemons[10], e.g. the daemon for the Android Debug Brigde (ADB).
3. After the daemons are started, Init starts the *Zygote* process.

[10] A daemon is a low level process sitting on top of the hardware abstraction layer and establishing communication with the hardware

4. Next, Init starts the runtime process.
5. The runtime process starts the *Service Manager* and registers this as the default service manager.
6. After the runtime finished the initialisation it sends a request to the *Zygote* to start the system server process.
7. Receiving the request, *Zygote* forks itself and starts the *System Server* as the first process running inside the DVM.
8. The *System Server* starts the *Audio and Surface Flinger* to be able to control the display and audio output device.
9. These register themselves at the *Service Manager* so they can be used by other higher level applications.
10. After this, the *System Server* starts the *Core and Hardware Services*.
11. These register themselves then to the *Service Manager* as well.

After this, the system is ready to execute the first activity. This activity is the home application, which shows the home screen. This is done by the activity manager, that sends a request to *Zygote* to start the home application.

4.3 Dalvik Virtual Machine

The DVM is the main part of the Android runtime environment. It was designed to fit the needs of a mobile device, which means it's capable of running on a slow CPU[11], little RAM[12] and doesn't necessarily need swap space[13]. Another constraint was the need to be able to run on battery power only, as this is the power source of all mobile devices, as of now. In order to meet these constraints, the DVM itself is optimized, as well as the compiled Java binary.(Google Inc., 2008b)

[11](Central Processing Unit), the main part of a computer that computes information and commands of applications (Elektronik-Kompendium.de, 2012a)
[12](Random Access Memory), memory to store the data of running applications in (Elektronik-Kompendium.de, 2012b)
[13]memory that is used when the RAM is full

4.3.1 Memory Optimizations

At the beginning of the development of the DVM it was specified that it should be able to run on any mobile device which offers at least 64MB of RAM. However, as the system kernel itself and the system services needs RAM as well, the DVM needs to be able to run with as less than 20MB of RAM. Additionally, due to the Android security model, which relies on process separation, meaning that each application is running as a separate process with its own address space and preventing applications to interact or interfere each other at memory level, the DVM needs to manage this as well. (Google Inc., 2008b) One step to reduce the needed memory was a new file format called .dex file. A .dex file is a Dalvik executable which consumes less memory than a standard Java .class file. This is achieved by storing multiple classes[14] inside a single .dex file and sharing identical values as opposed to having a .class file for each class with possible redundant data inside them.(Google Inc., 2008b) Another step that was taken was to share libraries between applications. This means, that a library is loaded only once in the memory of the device to be used by any other application that will be running. Following that, the DVM implements a copy-on-write policy, which means that as soon as an application writes something to shared data, the data is copied into the local application heap and modified there. This reduces not only memory consumption, but also time consumption as data is only copied when needed.(Google Inc., 2008b) The garbage collector[15] of the DVM is optimized as well. As the DVM shares as much data between processes as possible, the garbage collector must consider this. However, since every application is running as a separate process with separate heap[16] and in its own instance of a DVM, each app has its own, independent garbage collector that must respect sharing. To achieve this, the mark bits[17] of the objects in memory are stored separately instead of with the objects. (Google Inc., 2008b)

4.3.2 CPU Optimizations

To minimize CPU work time, an application is optimized when it's installed on the device. These optimizations include byte-swapping and padding, to improve the memory layout which allows faster memory access. Static linking is also done during install-time, which means that symbolic links to method may become a simple offset in a virtual function table[18], as well as

[14] an entity in object oriented programming
[15] an algorithm that manages memory in programming languages such as the Java programming language. It automatically deletes memory that isn't used anymore
[16] memory with a dynamic size used by an application
[17] used by a garbage collector to mark objects that are in use
[18] a table used by object oriented programming languages to determine which function has to be called in case of inheritance

inlining[19] native functions, so that function calls can be saved. Another optimizing is pruning of empty methods to avoid calling these.(Google Inc., 2008b)

4.4 Starting an Application

When the system needs to start an application (e.g. because the user of the device starts an application), the system first checks, whether this application is already running and just brings it to top if it is. Otherwise, the system needs to create a new process for the new application. In order to save performance, it's first checked if a unused process is already available, which is then used. When there's no unused process available, the system needs to create a new one, which is done using the "zygote process". After starting, every application is executed in its own user-space, with its own, unique username.

4.4.1 Zygote Process

As mentioned in the last section, the Android system starts an process called "zygote". This process is the DVM, which is part of the Android runtime and used to execute other applications. However, this process is a blank instance of the DVM, as it doesn't execute anything. Instead, it is used to created new instances of the DVM for execution by forking[20] the "zygote" process and loading the application data into the newly created process. (Google Inc., 2008a) The newly forked process shares its heap with the zygote process. This saves the system from reserving heap space for an application that doesn't write data to the heap or alter the data on it. As soon as this happens, the heap is copied and the new process gets his own heap, on which the application can write and modify data. Since the memory pages are directly copied from the zygote process, the memory layout remains the same. (Google Inc., 2008a)

4.5 Binder

The Android system provides an interface called binder interface, which is used to establish IPC between applications. This interface is a low level implementation inside the kernel space[21] of the Android system. The Binder interface used in the Android system origins from the OpenBinder project, which runs under Linux to extend the existing IPC[22].The

[19] an optimisation where the call to a function is replaced by the code of the function that would have been called, thus saving the actual function call
[20] creation of a new process using the fork() system call
[21] memory area in which the kernel and its modules are operating, opposite to user-space
[22] (Interprocess Communication), a method to exchange data between processes (Wikipedia, 2012j)

Binder framework consists of 3 layers. The highest layer represents the API[23] for Android applications and is located inside the Java layer of the Android system. The next layer is a middleware, which is the user-space implementation and the lowest layer is a driver in kernel spaceSchreiber (2011).

4.5.1 AIDL

The AIDL allows developers to define a programming interface that both, the client and the service, agree upon to communicate with each other to using IPC. Since processes cannot access the memory of other processes, the data must be decomposed into data the glsOS can understand, such as primitive data types, and be marshalled by it to the other process. To simplify this, the AIDL is used to generate the necessary code to handle the marshalling.

4.5.2 Java API

As the top layer, the Java API offers the features of the Binder to the applications inside the DVM. The kernel itself, as mentioned in the last section, cannot handle Java class objects, which have to be decomposed. In order to achieve this, the Binder framework provides a Java interface called Parcelable. Any Java class that has to be send through a Binder interface has to implement this interface. When data is send through a Binder, this is called a transaction. During a transaction, the data sent from one application to another, the data is composed inside parcels. The Parcelable interface requires the implementation of a function and a CREATOR interface. The function is used when an object of that class needs to be decomposed into primitive data. This data is then, also by this function, saved inside the parcel, so the object can be sent to the receiver. When the receiver receives the parcel and wants to retrieve the sent object from it, the CREATOR interface of the objects class is called. This interface has to be implement to retrieve the primitive data from the parcel an build the Java object from it.

4.5.3 Middleware

The user-space facilities are implemented inside the middleware using the C++ programming language. This middleware is used by the Java API through the JNI. The middleware implements the marshalling and unmarshalling for transforming object information to a parcel of data. In Addition, the middleware also implements interaction with the Binder kernel driver and the shared memory Schreiber (2011). Since the middleware is implemented in C++,

[23] (Application Programming Interface), a specification intended to be used by software to communicate with each other or to provide services to other applications (Wikipedia, 2012b)

programmers can also use the Binder interface when they are using JNI instead of pure Java to create their applications. However, they have to relinquish features of the Java API layer in that case.

5 Implementation of the Security Architecture

This chapter covers the security architecture in detail as well as its sample implementation. At first, the security architecture will be introduced, with its functionality and the work-flow of the components in it. Following that, an implementation of this architecture is shown with the modifications that were made to any kernel modules and coverage of new modules added to the kernel.

5.1 Security Architecture

The security architecture that will be proposed in this section follows closely the assumptions made in chapter 3. A MTM, that were introduced in chapter 2, will cover the hardware requirements of the architecture. Even though no MTM has been manufactured as of now, it can be expected that MTM hardware implementations will possible emerge in the near future. As discussed in chapter 2, the security architecture relies on a chain of trust which is established when the device starts. This makes a secure boot necessary as this ensures that the hardware as well as software was not tampered with and the device will be in a well-known trustworthy state after the boot. This requires the MTM to be set up in to MRTM mode at manufacturing time, as a MRTM performs a secure boot procedure when the device is activated. In addition, a MRTM allows to sign RIM certificates, that are used to control the execution of applications. The secure boot procedure of the MRTM meets the requirement of the secure startup mentioned in chapter 2. However, as this check only occurs at the activation of the device, any threats and manipulations happening during runtime will not be detected by the MTM until the next boot. As a mobile device has only limited computational power and its power source is usually a battery installed in the device, a continuous scan for threats is not suitable. Establishing a black-list[1] to mark malicious software and block the execution of it when the software is started would avoid having a continuous scan. However,

[1] a list of entries indicating not accepted entries, contrary to a white-list

as new malicious software arises, the black-list will have to be updated every time. In addition, when the black-list is corrupted in any way or missing, the security architecture would be nullified. Therefore, a white-list approach is chosen to be used for verification. Only if a application is listed in the white-list it will be allowed to be execute. Any other software will be prevented from execution. The white-list consists of entries which in turn consists of a RIM certificate of the corresponding software signed by the MRTM on the device. This white-list represents a trusted application list TAPL[2] that is maintained by the administrator of the mobile device (Landsmann, 2011). The device administrator can request the MRTM to create a RIM certificate for a software using previously calculated hash which is then used as reference. This corresponds to the security architecture proposed in chapter 2. This reference hash is used to verify the software before it is started on the device. The TAPL is looked up to find the corresponding RIM certificate in it, when the certificate is found, the hash of the software is calculated at load-time and compared to the reference hash of the certificate. Only if the hash value matches the reference value, the software is allowed to be executed. As shown in chapter 2, the MRTM needs to be extended to perform this runtime verification to keep the chain of trust valid as well as the OS, in this case the Android OS, has to support the MRTM attestation abilities. This means that every execution request of a software needs to hook up an attestation. As every execution of an application leads to the creation of a new process (cf. Chapter 4), the attestation is done at this point. Every process is also created as a child of the calling process. When a new process is created, not only the certificate of the program that will be executed is verified, but the calling process is verified by another function as well. This prevents applications that slip through the first attestation from creating child processes. Additionally, to avoid uncertified software from being executed, the scheduler verifies a process before giving resources to it. Chapter 4 introduced the Android OS as based on the Linux kernel with modifications for Android and mobile devices. It was also stated, that Android applications are running inside the DVM, by creating a new process, created by the kernel, so that every application is running in its own process. The process creation is secured by providing attestation functions inside the kernel and inside the DVM. Both attestation point are secured by the secure boot procedure of the MTM, as this would detect an manipulations that would allow bypassing the verification. The verification, attestation and management for the security architecture are mainly implemented as kernel modules. These modules also provide an interface to interact with them. This is necessary because the applications started inside the DVM are not handled by the kernel, but by the Android OS by providing functions to load and start an application inside the DVM. To secure this, the DVM

[2](Trusted Application List), a list that contains trusted applications that are allowed to run on the device

is extended with functions which interact with the kernel modules when an application is to be started. Creation of native processes are still fully handled be the kernel and directly verified by the security architecture modules. This allows the security architecture to protect all possible process creations, whether natively or through the DVM, on the Android OS. (Landsmann, 2011)

5.1.1 Architecture Modules

The introduced security architecture is implemented based on Google Inc.s Android OS in the version 2.3, commonly known as "Gingerbread". The following figure shows the modules of security architecture as well as their interaction inside the OS.

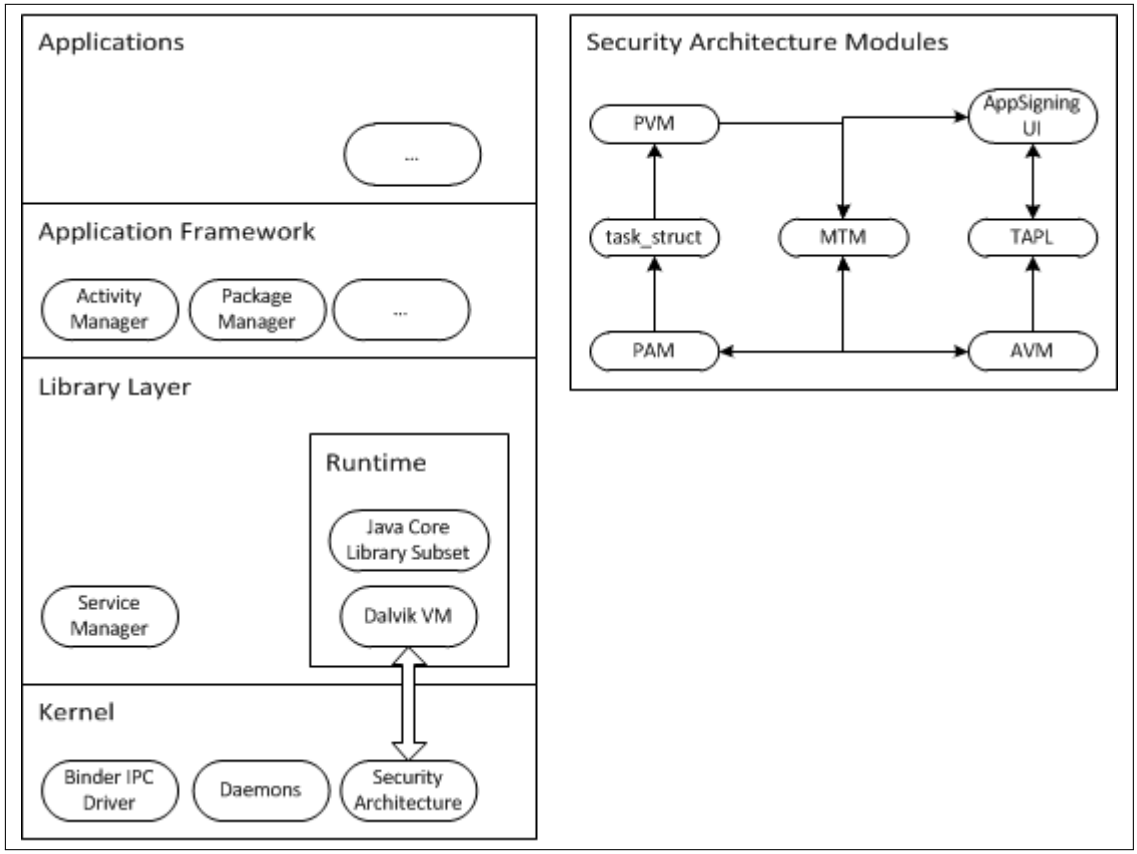

Figure 5.1: Security architecture modules location and interaction (Landsmann, 2011)

Security Architecture Keys

This section shows the keys and security primitives the security architecture relies on as described in (Landsmann, 2011):

- $Key_{MTMAuth}$: In the proposed security architecture, this key is used to authenticate requests from an administrator of the MTM.
- Key_{HMAC}: This key is used for authentication and verification by PAM and PVM.
- Key_{PubSig}: This key is used by the AVM to request the MTM to verify a hash against the signed TAPL entries.
- $Key_{PrivSig}$: This key is sealed inside the MTM and never released as it's exclusively usable and accessible by it. The MTM uses this key to sign a given hash value.

MRTM Wrapper

During creation of this thesis, no MTM hardware implementation is currently available. To decouple the security architecture from a real MTM hardware, a kernel module was implemented. This module acts as a substitute for a real MRTM and provides the necessary interfaces to the security architecture. The kernel loads this module at power-on together with the rest of the OS. This substitution fits the requirements made in chapter 3 as it makes the keys from the key hierarchy accessible within the kernel and provides cryptographic functions needed for the signing and verification mechanisms. (Landsmann, 2011)

Trusted App List

The TAPL contains the RIM certificates of the trusted applications. This module provides function to interact with this list, such as adding and removing RIM certificates from it and handing out the TAPL to a requesting entity like a kernel module.

Application Verification Module

This modules verifies the computed hash value of the software against the RIM certificates contained in the TAPL. For this, the MTM requested to verify the calculated hash value against the RIM certificates. The module then returns an error code in the case of verification failure, otherwise a success code is returned.

Process Authentication Module

The PAM computes the HMAC using the PID of the process. This is done by using a secret key Key_{HMAC} stored inside the MTM.

Process Verification Module

The HMAC computed by the PAM of the process is used by this module to verify the process. This is done by computing the HMAC of the process again, in the same fashion as done in the PAM. If both HMACs match, the process is allowed to be scheduled and to create child processes.

User Interface

The user interface provides the features of the security architecture to the administrator. It passes signing request containing the hash of a program and the authentication key of the administrator to the MTM. If the authentication was successful, the MTM returns a RIM certificate which is added to the TAPL. It's also capable of managing request targeted at the TAPL.

5.1.2 System Integration

The integration of this security architecture into the system is based on four steps, the creation of new processes, starting of new applications, the boot sequence and, to allow the administrator to interact with the architecture, an UI. This integration is shown here as it's described in (Landsmann, 2011).

Boot Sequence

After the MTM has successfully finished the secure boot procedure up to the state 8 in the figure above, the kernel modules of the security architecture are loaded into the kernel. The kernel then starts the INIT process, which calls the *do_fork()* function. The PVM is hooked into this function to verify processes before allowing them to be forked. As the INIT process is the first process being created and its PID is 1, the PVM can identify this process and skip the verification. The PAM then computes and endorses the INIT process with a HMAC and returns a success code. After the INIT process is started, no verification is skipped to ensure the trustworthiness of the device. The OS repeats this procedure until it's fully loaded. (Landsmann, 2011)

Process Creation

As mentioned in chapter 4, process creation is done using the *fork()* system call. This function calls the *do_fork()* function for the actual process creation. The PVM is hooked into the function to verify the creation of new processes. For this. it requests the Key_{HMAC} from the MTM and computes an actual HMAC from the PID of the process using this key. The module then compares the computed HMAC with the endorsed HMAC. If both values match, a new process is created. The PAM now requests the Key_{HMAC} from the MTM as well. It's used to compute a HMAC for the newly created child process and endorse it with it. (Landsmann, 2011)

Starting a Software

Chapter 3 shows that the start of a software is handled different by Android compared to Linux. To further ensure the trustworthiness of the device, the security architecture has to be hooked into both mechanisms.

Linux After a new process is created according to the mechanism described in the last section, the *exec()* system-call is called. At this point, the binary of the application is used to calculate a hash value and the AVM requests the Key_{PubSig} from the MTM as well as the TAPL from the TAPL module. The MTM is then called to verify each entry using the previously requested key. If the verification is successful, meaning the TAPL contains an entry for the application, the *exec()* call is continued, resulting in starting the software. Otherwise the system-call is aborted resulting in termination of the execution. (Landsmann, 2011)

Android An Android application is executed inside a DVM. A new DVM process is created as described in the previous section. The DVM then receives the information about the application that is started. Similar to the native start of an application in Linux, a hash of the binary is calculated inside the DVM and passed to the AVM. The AVM then again requests the Key_{PubSig} from the MTM and the TAPL from the TAPL module. The MTM is then called to verify each entry using the previously requested key. If the verification is successful, the DVM is allowed to execute the application, otherwise the start is terminated. (Landsmann, 2011)

Administration The administration is done using the character device provided by the architecture. This device passes requests from the administrator, when the $Key_{MTMAuth}$ is used, to the AppSigningUI module. This module calls the MTM to verify the administrator using the

$Key_{MTMAuth}$. On a successful verification, the AppsSigningUI module continues processing the request. A request can be one of the following:

- Receiving the TAPL
- Signing and adding the signed value to the TAPL
- Removing a signed value from the TAPL

If the authentication fails, the AppSigningUI passes the error code to the character device which the administrator can receive it from. (Landsmann, 2011)

5.2 Implementation

Martin Landsmann implemented the security architecture described in the last section. His implementation closely follows the modules presented previously and partly ties up with (Ugus und Westhoff, 2011). The implementation is targeted at Android OS version 2.3.3, commonly known as "Gingerbread", running on a Linux kernel version 2.6.35. He modified the DVM to support the security architecture as well as the kernel. These modifications will be covered in this section. In addition, several new kernel modules were added to implement the security architecture. (Landsmann, 2011)

5.2.1 Security Architecture Modules

To implement the security architecture, Martin Landsmann implemented several modules resembling the modules described in the previous section. These kernel modules are introduced here.

MTM Module

This module represents the interface between the security architecture and a real hardware MTM. Upon loading, the module first performs an integrity check if the OS and compares it with the previously MTM computed values by asking it to handout the corresponding RIM. Once the check is successful, the module retrieves the keys from the MTM and is able to provide it functions, letting the MTM sign and verify SHA256 values, to the other modules. Otherwise, if the check fails, this module unloads itself. As there is currently no MTM hardware available in mobile devices, these checks are skipped and assumed to be successful.

Key Storage Module

The Keys Key_{PubSig} and Key_{HMAC} are provided to the security architecture by this module. When this module is loaded, it requests the MTM Module to reveal the keys and later arbitrates these keys to the other modules.

TAPL Module

The TAPL module manages all TAPL entries. Each entry consists of a signed SHA256 value and the filename of the corresponding binary file. As no MTM hardware is available yet, no genuine RIM certificates can be used. This module also provides the interface to interact with the TAPL.

Autoload Module

This module provides the TAPL module with the information of the stored TAPL on the device. It reads the file containing the information and uses the interface provided by the TAPL module to add the entries to the TAPL.

AVM Module

This module provides the verification of a given SHA256 value. The key storage module is requested to handout the Key_{PubSig} and the TAPL is requested from the TAPL module. The SHA256 values contained in the TAPL are then verified against the given SHA256 value using the MTM module. After the verification, the result is returned to the caller.

UI Module

This module provides an interface for the administrator to interact with the security architecture. Communication between the security architecture and the administrator is done by connecting to the character file */dev/sec_device* and using the *ioctl(...)* system call. Requests are marshalled into a special format and is send to the character device. This device unmarshalles the request and returns the result in the same way. This module supports commands to:

1. sign a SHA256 value and add it to the TAPL
2. remove a entry from the TAPL
3. receive the content of the TAPL

These commands are identified by command numbers also passed to the character device.

Verify Bridge Module

5.2.2 Kernel Modifications

To implement the security architecture, the kernel has to support the features provided by the previously presented modules. This was done by Martin Landsmann with kernel modification which will be shown in this section. (Landsmann, 2011)

Process Creation

The creation of new process has been modified in a way that the PAM and PVM modules are used by the *fork()* system call. These modules are used to perform authentication and verification operations. This new behaviour is illustrated in the following figure and described with the bullet points 1 to 7.

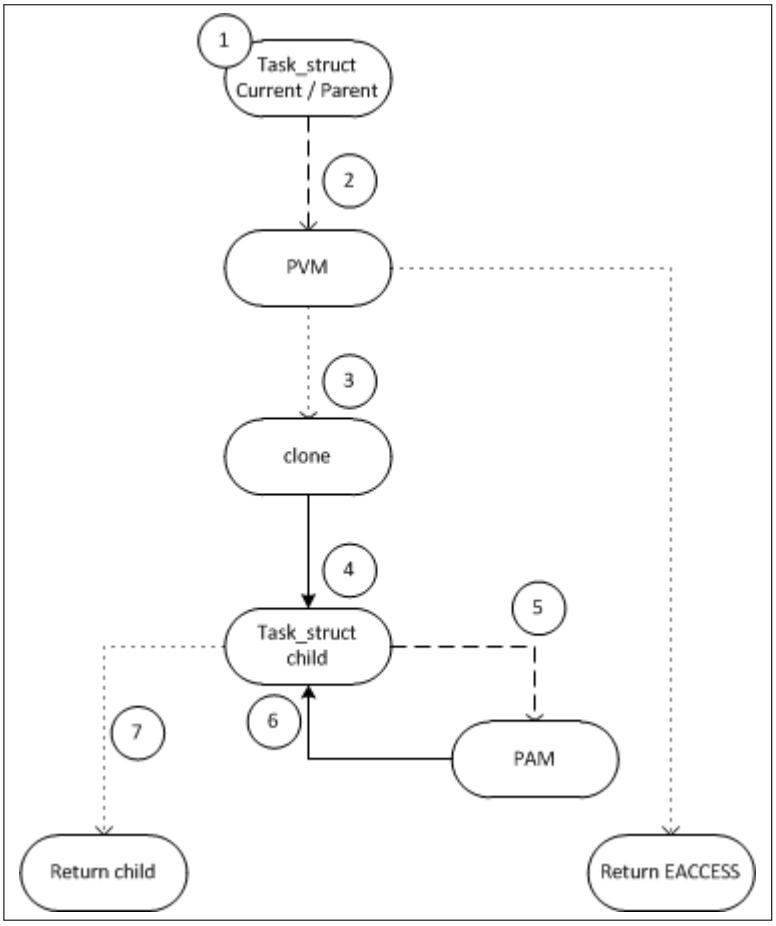

Figure 5.2: Visualisation of the Process Creation (Landsmann, 2011)

1. The process creation is invoked by a process, the creation is performed inside the *do_fork()* function
2. Before the caller process is cloned, the HMAC of the calling process is verified by the PVM
3. The *do_fork()* function continues when the verification is successful
4. The child process is cloned
5. The HMAC for the child process is computed by the PAM
6. This HMAC is stored inside the child process' *task_struct*
7. The child process is successfully created and returned by the *fork()* system call

If the verification of the parent process' HMAC fails, the function returns with *-EACCESS* error code and the creation of the child process is terminated. This procedure is done for all process creations except the *INIT* process. This process is verified by the secure boot and is assumed to be trustworthy automatically. However, as with the other processes, the HMAC is still computed and stored inside the *task_struct*. (Landsmann, 2011)

Binary Execution

The kernel was extended to calculate a SHA256 value of the binary inside the *exec()* system call. This value is then passed to the AVM for verification. If the verification is successful, the execution of the binary is continued, otherwise it's terminated with an error code. (Landsmann, 2011)

5.2.3 DVM Modifications

The DVM was also extended with a verification function. When the DVM loads a binary, it's SHA256 value is computed and passed through the verification bridge to the security architecture. The security architecture then verifies the SHA256 value utilising the AVM. At a successful execution the application is executed as normal. When the verification fails, an exception is thrown within the DVM, which terminates the loading of the application. (Landsmann, 2011)

6 Implementation of the Context Recognition

The security architecture introduced in the last chapter is now extended to be context sensitive. This chapter shows the architecture as well as the implementation of this extension. The implementation also covers further modifications done to the modules created for the basic security architecture.

6.1 Security Architecture

The security architecture proposed in this chapter extends the architecture described in the last chapter with an element to act in regard of defined contexts. To ensure the most flexibility, a context may be defined in any suitable way with the only restriction that it must be recognisable, e.g. a wireless network or a gps location. This mechanism, the recognition of contexts and acting upon it, is decoupled from the rest of the security architecture. This provides advantages such as being able to easily change the underlying security architecture. The main advantages however is, that a corrupted context recognition does not affect the security architecture it is based upon. The context sensitivity is provided by multiple TAPLs. These TAPLs are managed by the security architecture, the context recognition cannot alter them in any way, it is not able to add or remove entries from any TAPL stored in the security architecture. This is done by a management application that provides this functionality to the user, as described in the previous chapter. This extension to the architecture is only capable of managing contexts and switching the current active TAPL in case of a context change. Similar to the security architecture, the architecture of the context recognition provides an interface to manage the contexts. The device administrator can send requests to the architecture to:

- Add a context
- Remove a context
- Retrieve all contexts

- Retrieve the currently active context

The device administrator can create an arbitrary amount of contexts and add them to the context recognition. Additionally, he is able to create an arbitrary amount of TAPLs as well. The amount of created TAPLs and contexts does not have to be equal. A context can then be connected with a TAPL and, upon recognition, the connected TAPL will be set as active. This has the effect, that only applications listed in the currently active TAPL will be able to start. This mechanism is provided by the security architecture. In addition, when the TAPL is switched, the architecture checks if any application that is running needs to be terminated. This is the case when the application is not listed in the TAPL, as this means that the software is not allowed to run in this context. This architecture can be implemented in many ways as low-level access is only needed to communicate with the kernel. Depending on how the architecture is implemented, advantages and disadvantages are gained. Implementing the architecture as a low-level software, e.g. a kernel module, can improve the performance and security. However, the implementation can be harder to maintain and extend. On the other hand, implementing this on a higher level might make it easier to bypass the architecture, but also easier to maintain and expand.

6.2 Architecture Modules

Similar to the security architecture, the context recognition is based upon Google Inc.s Android OS version 2.3. The context recognition consists of mulitple modules which are introduced and described in this section as well as their communication between each other and the rest of the security architecture.

6.2.1 Context Recognition Service

The context recognition service is the central part of this architecture. It communicates with the rest of the security architecture in case of contexts changes and loads up the context recognition itself as well. Upon startup, the service reads a file containing the context data stored on the mobile device and loads the contents of it. The loaded contexts from this file will be added to context observers. This file is only read on startup, once loaded, the service only saves new contexts to the file, always completely overwriting it. After the initial startup is finished, the service waits for notification from the context observers that a new context is active and switch the TAPL. This service is the only part of the context recognition that will ever communicate the security architecture implemented in the kernel.

6.2.2 Context Observer

A context observer is a module to observe specifically one type of context. These modules are independent from each other and from the service. An arbitrary amount of contexts can be added to an observer of the matching type, e.g. a context for wireless networks is added to an observer for these contexts, not to an observer of a different kind. Once the observer recognises one context to be active, it notifies this context about it state change. Additionally, a context observer may be implemented in any way that seems suitable fitting for the use case. The only restriction is that is has to inform the corresponding contexts about state changes.

6.2.3 Context

A context represents an abstract definition of any environmental information the mobile device can be, or is, in. These may be based on any information that the mobile device can collect or otherwise compute, e.g. a wireless network or the current GPS location. A context may only be added to a matching observer. Furthermore, a context is always contained in at least one context group. Once the context is notified by the observer that its active state has changed, it delegates this notification to every context group it is contained in.

6.2.4 Context Group

An arbitrary amount of contexts can be grouped to a single entity inside this architecture. This offers the possibility to define a context that is only active, when multiple sub-contexts are active. A use case for this might be a location at a certain time. Instead of defining a new context with a new observer fitting only this requirement, it is possible to group a location context and a time context to a single entity. To keep the communication between the modules simple without any exceptions, a context which can be represented by a single information is stored inside a context group as well. When a context group receives a notification from one the contexts contained by it, it is checked if the context group is active by receiving the active state of all context inside this group. Only when all contexts are in an active state, the group is active as well. Once the group receives a notification that a single context inside this group has became inactive, the whole group becomes inactive. The context group also contains the id of the TAPL it is bound to and will become active when the context group becomes active.

6.2.5 User Interface

As the contexts and contexts groups are stored inside the architecture, an interface is needed to manage these. Only the device administrator is able to do this, as the security architecture

can be corrupted when an attacker changes the data inside the architecture. The user interface provides the following functionality:

- Adding of contexts
- Removing of contexts
- Creating context groups
- Deletion of context groups
- Adding contexts to groups
- Removing contexts from groups
- Retrieve all contexts
- Retrieve current active context
- Save data
- Authorising
- Deauthorising

This allows the device administrator to control and manage the contexts and groups inside the architecture. The user interface however does not expose the original data stored inside the architecture to the administrator, it always creates copies which are then passed through. In addition, incoming data, such as new contexts, are also copied. This decouples the architecture from external changes due to exposing or receiving original data.

6.3 Implementation

The Implementation of the previous introduced concept is splitted into two parts. One part covers the necessary work done in the kernel modules, such as preventing unwanted applications to be executed by the system, the other part covers the CRS, which is observing the environment for possible context changes. This example implementation is targeted at the Android OS version 4.0, which means, that the kernel modules are implemented using the C programming language, whereas the CRS is implemented using the Java programming language. This figure shows the communication inside the CRS and to other applications. The two separated boxes visualise two separate application on the same mobile device. The

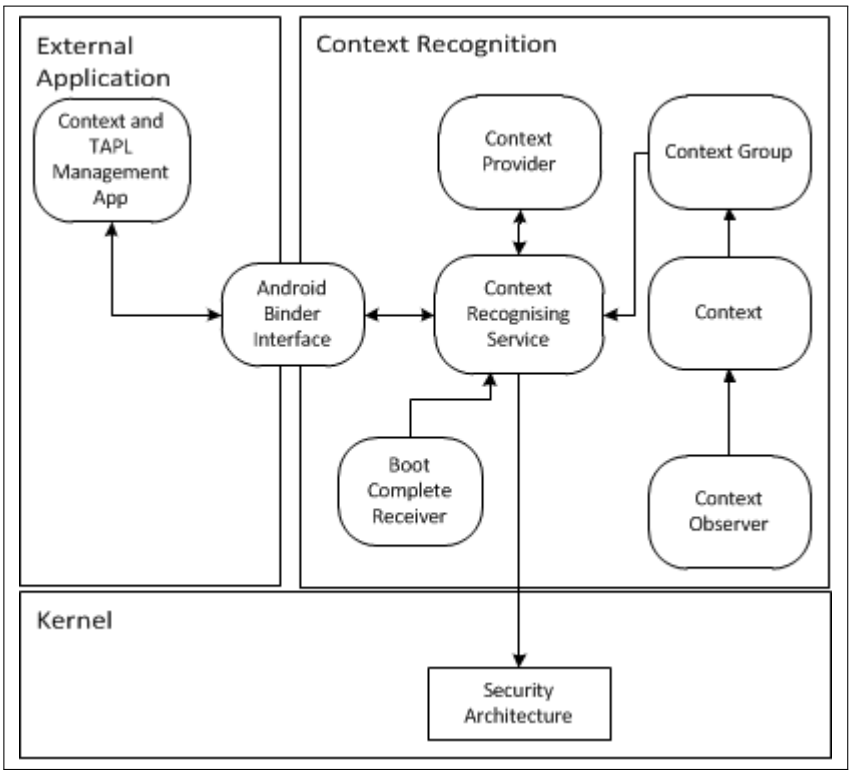

Figure 6.1: The communication model

right application is the implementation of the CRS, the left box stands for any application that needs to communicate with it. As shown, the communication to other applications goes through the binder interface, which offers several features for extern applications to interact with the CRS. The core of the service communicates with context observers, which report back as soon as a context change has happened to inform the service about this. An observer itself doesn't communicate with other observers. When the core receives an update from an observer, it sends a broadcast intent to the android system to inform extern application that the context changed. The decision to implement the architecture at a higher level, inside the Android runtime was made due to maintainability reason. A implementation like this can be easily upgraded and patched over the air. If the implementation would have made using kernel modules, the mobile device would have to be flashed with a new image each time an update needs to be installed.

6.4 Context Recognising Service

This service is implement by using modules for observing the enviroment and ensuring communication to other applications using the Android binder interface. The service itself is implemented as an Android service and thus runs as a background process. It is started after the device finished its boot procedure and initialises itself and loads a specific file from its directory. During the initialisation, all available context observers are added to an internal list. The file, which is loaded next, contains the information of all contexts, which were added for observing to the service. The contexts are loading one after another and are added to the correct observer, when a fitting observer was found during initialisation. If none observer for a context is found, the service prints a warning to the ADB and ignores the context. During runtime of the service, a management application may connect to it to add new contexts, get all contexts or get the current active context. This is done using the binder interface.

6.4.1 Context

Contexts are used to define certain environmental states, such as existing wifi-networks, GPS coordinates or times. This example implementation implements two different contexts:

- A context representing wireless networks, called *WifiContext*
- A context representing a time-frame, called *TimeContext*

The *WifiContext* represents a wireless network by its broadcasted SSID and is valid when the mobile device is within the range of a wireless network with the specified SSID. The *TimeContext* uses a start time and an end time to represent a time-frame. The context is active in between these two times. A special case of a context is the implementation of a common context. This is a context, that is always valid and is added to the service as soon as it's launched, either by an external application or by the boot complete receiver. It was implemented in order to have always at least one valid context, even if there aren't any other contexts in the server or if none other context is valid at the moment. This eliminates the requirement to handle the special case of having no valid contexts. However, as the common context is always valid, it can not be sent through the Binder interface, as this would allow extern applications to create situations in which a context, that would allow unwanted applications to run, would be valid.

6.4.2 Context Group

To establish a possible logical AND relation between any given number of different contexts, context groups are able to group contexts to a single semantic entity. Context groups, like contexts, are identified with an unique name and can hold an arbitrarily number of contexts. A context will, as soon as the respective context observer changes their valid state, notify all groups it is in. The group will then check to see if the other contexts inside the group are valid as well and if they are, it will set it's own valid state to true and notify the service about this. A context group is only valid when all the contexts it contains are valid as well, as long as there is at least one invalid context, the whole group will be invalid. Context groups also contain the TAPLID, which is used to identify the TAPL inside the low level libraries of the implementation. Contexts which shall not stay in a logical AND relation with other contexts, are added to groups that only contain that single context. An example of this is the common context. As already stated, that context is used to have a valid context at any given time. At the start of the service, the common context is added to a context group as its only element. This context group thus is always valid. In addition, this special context group has a special TAPLID that allows the mobile device to run normally without any restrictions.

6.4.3 Context Observer

A context observer provides an abstract interface to the CRS, the purpose of this class is to observe the environment and check if a context is entered or left. Upon start of the service, all available context observers are loaded and will receive contexts to observe. Every implementation of a context observer has to expose the kind of context it is able to observe, so that the CRS can choose the correct observer for a context. The main part of a context observer, the observing of contexts, can be implement independ from the rest of the service or other context observers.

6.4.4 Context Provider

When the context recognising service loads the file containing the context and context group data, this file is loaded per line. Each line is describes one element that needs to be created. The context recognising service passes the line to the context provider which extracts the necessary information from it. Through reflection, the class representing the context or context group is found and the describing elements are passed to the constructer of this class. The constructor then creates the class using the description from the context provider. This simplifies adding new contexts to the implementation as it reduces the amount of work needed to do this. When

a new context needs to be added, it only has to be able to store a text-based description of it inside a file and provide a constructor that is able to recreate the context from this description. In addition, a context observer needs to be added for this context.

6.4.5 Boot Complete Receiver

The boot complete receiver is an essential part of the service. As shown earlier, the Android system sends various broadcast intents through the system, which can be received by applications using broadcast receivers. The service comes with a broadcast receiver, which waits for the BOOT_COMPLETED broadcast intent sent by the system after the device finished its booting procedure. Upon receiving this intent, the boot complete receiver starts the CRS. As of version 4.0 of the Android system. the behaviour of receiving the BOOT_COMPLETED broadcast intent has changed. Prior to version 4.0, this intent was always received by an broadcast receiver that was registered for it. This however, made it possible for malware to be started at system start without notice by the user. To prevent this, all applications newly installed on the device are in the internal pause status, which is only changed when the user starts the application on its own. The BOOT_COMPLETED intent however, is not received by broadcast receivers which belong to an application that is in the pause status. This causes the boot complete receiver of the service to fail by not receiving the broadcast intent unless the service was started by the user once. This doesn't need to be done by the user directly, instead it is also possible that an application, that was started by the user, starts the service.

6.4.6 Binder Interface

The Android system provides various ways to implement IPC, the binder interface is a low level implementations of this. The CRS uses this interface to establish a communication with an external application, such as an management application. Through this interface, it's possible to add new contexts and context groups to the service as well as remove existing contexts and context groups from the service, receive all currently added contexts and groups from the service and receive the current active context group from the CRS as well as adding or removing contexts to or from groups. The modifing requests require the device adminstrator to be authorised before being executed. To avoid exposing the original inside the service and thus making modifications possible, this interface only returns copies of the data, never the original hold in the service. In addition, an application that wants to connect with the CRS during its execution time has to ask the user for permission to do this before installing. This makes it possible to warn the user that the application that is about to be installed, may connect to the

service and add or remove contexts. If that permission is not granted, an exception is thrown by the Android system.

6.4.7 Context

Contexts are used to define certain environmental states, such as existing wifi-networks, GPS coordinates or times. The example implementations demonstrates the use of such a context class by implementing wifi-contexts and time-contexts. Wifi-contexts are recognized by the SSID broadcasted by wifi-networks in the surroundings and is valid as soon as a wifi-network with the specified SSID is found, whereas time-contexts are defined with a start-time and an end-time in which between the context is valid. A context has a unique name, which is used to distinct different contexts and is also used by the Binder interface to retrieve a context from the service. A special case of a context is the implementation of a common context. This is a context, that is always valid and is added to the service as soon as it's launched, either by an external application or by the boot complete receiver. It was implemented in order to have always at least one valid context, even if there aren't any other contexts in the server or if none other context is valid at the moment. This eliminates the requirement to handle the special case of having no valid contexts. However, as the common context is always valid, it can not be sent through the Binder interface, as this would allow extern applications to create situations in which a context, that would allow unwanted applications to run, would be valid.

6.4.8 Context Group

To establish a possible logical AND relation between any given number of different contexts, context groups are able to group contexts to a single semantic entity. Context groups, like contexts, are identified with an unique name and can hold an arbitrarily number of contexts. A context will, as soon as the respective context observer changes their valid state, notify all groups it is in. The group will then check to see if the other contexts inside the group are valid as well and if they are, it will set its own valid state to true and notify the service about this. A context group is only valid when all the contexts it contains are valid as well, as long as there is at least one invalid context, the whole group will be invalid. Context groups also contain the TAPLID, which is used to identify the TAPL inside the low level libraries of the implementation. Contexts which shall not stay in a logical AND relation with other contexts, are added to groups that only contain that single context. An example of this is the common context. As already stated, that context is used to have a valid context at any given time. At the start of the service, the common context is added to a context group as its only element.

This context group thus is always valid. In addition, this special context group has a special TAPLID that allows the mobile device to run normally without any restrictions.

6.4.9 Context Observer

A context observer provides an abstract interface to the CRS, the purpose of this class is to observe the environment and check if a context is entered or left. Upon start of the service, all available context observers are loaded and will receive contexts to observe. Every implementation of a context observer has to expose the kind of context it is able to observe, so that the CRS can choose the correct observer for a context. The main part of a context observer, the observing of contexts, can be implement independent from the rest of the service or other context observers.

6.5 Library Implementation

This part covers the implementation to control the execution of applications on the mobile device. When a context change happens, the service notices this and calls the low level library implementation to handle this context change. The id of the now valid TAPL is passed as a parameter from the service. The low level implementation then sends a message to the kernel module that handles the current set of TAPLs with the new id received from the service. Upon receiving such a message, the module switches the current TAPL, which controls the execution of applications that will be started in the future. However, as it can occur that application that are prevented from executing are already running, a check is necessary to find these applications. If a blocked application is running at that time, the library tries to terminate it gracefully by sending it the Linux term signal. This is done to prevent data loss due to ungracefully deleting a application from the process list before it can persist any mandatory data from the system RAM.

6.6 Module Modifications

To make TAPL switches possible, the corresponding kernel module in the security architecture needed to be modified. These modification consist of adding functionality to show the content of specific TAPL by providing an id to identify this TAPL as well as add and remove entries from a specific TAPL in the same manner. Additionally, functions were added to create new, empty TAPLs. In this case, the module passes the id of the new TAPL back to the user interface. Finally, the module is also able to switch the current active TAPL to a new one and to terminate

unallowed applications in this case. Additionally, a TAPL is introduced that is bound to the common context inside the context recognition architecture. Therefore, this TAPL is active, when the device is in no defined context or the context recognition is not being executed. This is a fallback mechanism that allows the device administrator to define what applications are allowed to be executed when no context is defined, the device is in no defined context or when the context recognition is not being executed by the system because it has been corrupted by an attacker.

6.7 Workflow

This section describes the workflow of the context recognition in detail. At first, the workflow for automatically starting the context recognition is shown, secondly the context management and finally the context recognition is described.

6.7.1 Start of the architecture

The start of the application is done after the device finished the secure boot procedure and the boot procedure by the Android device. As the first steps were already described in the corresponding chapters, they will be skipped here. Figure 6.2 visualises the workflow of the start procedure.

1. After the Android OS has finished it's boot procedure as described in chapter 3, the system sends a *BOOT_COMPLETE* broadcast intent. This intent is received by the *Boot Complete Receiver*.
2. Upon receiving the *BOOT_COMPLETE* intent, the *Context Recognising Service* is started by the receiver.
3. The *Context Recognising Service* initialises itself by first loading the available *Context Observer* implementation. Following that, the *Context Recognising Service* reads the stored context data.
4. This data is parsed by and passed to the *Context Provider*, which recreates the stored *Contexts* and *Context Groups*.
5. Finally the *Context Recognising Service* adds the *Contexts* created by the *Context Provider* to the corresponding *Context Observer*

With these five steps the context recognition is set up and already starts observing the state of the contexts.

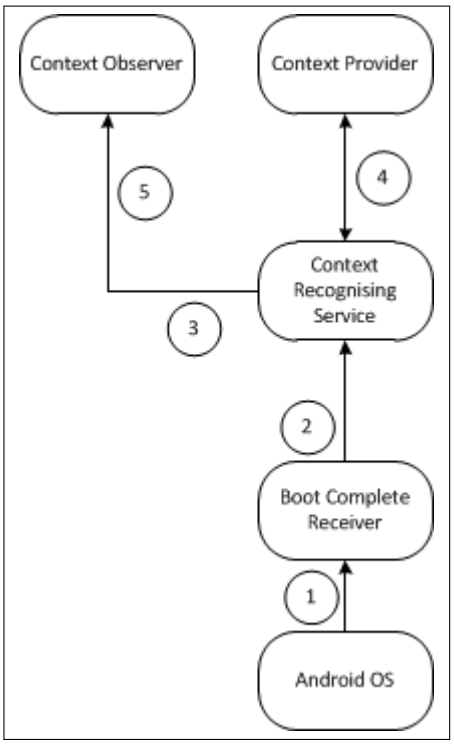

Figure 6.2: Workflow of the architecture start

6.7.2 Context Management

The device administrator is able to manage the contexts and contexts groups by using a management application that connects to the service through the provided binder interface. The communication is described in this section.

1. The *Management Application* connects to the provided *Binder Interface*. When the connection is established, the application can send new created contexts or context groups to this. When a new *Context* should be added to the service, the *Context* has to be preconstructed by the *Management Application*. This simplifies the procedure as a *Context* can be very complex. A *Context Group* however is only identified by its name, so it's sufficient to send this to the service.

2. The request is now send to the *Binder Interface*. A request may be any of the operations introduced in section 6.2.5., providing the necessary data.

3. If the *Binder Interface* receives a request to add a new *Context* to the service, it recreates a new, independent *Context* from the data provided by the *Management Application*.

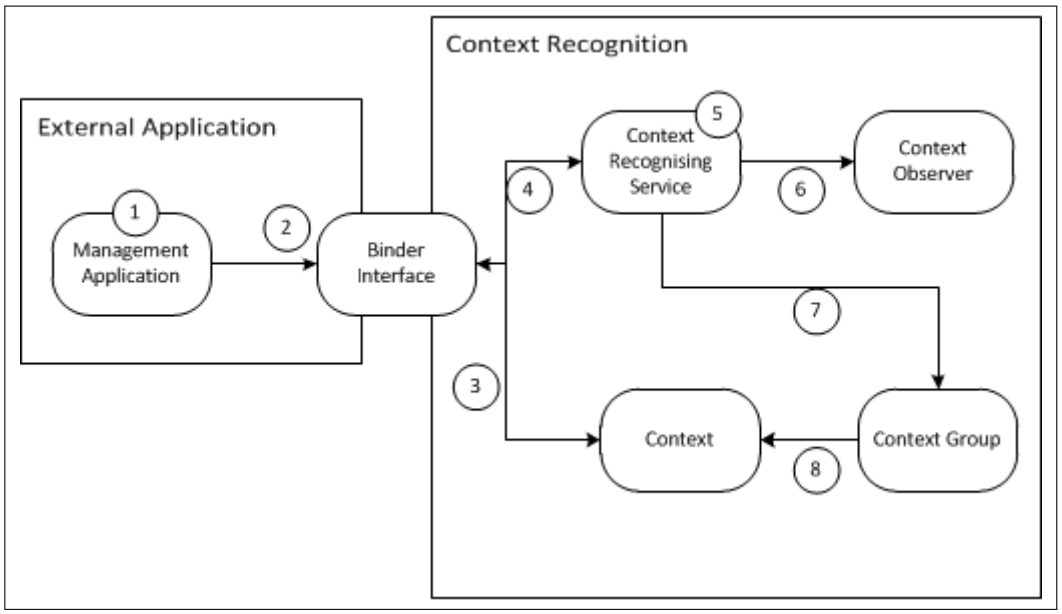

Figure 6.3: Workflow of the context management

4. The received data, e.g. a new *Context* or a *Context Group* are now passed to the *Context Recognition Service*, according to the request.

5. The *Context Recognising Service* processes the request coming through the *Binder Interface*. If the request is to receive data, such as an *Context*, the request is handled by the service directly. It sends the requested data through the *Binder Interface* in the same manner as the *Management Application* does.

6. If the request was to either add or remove a *Context* from observing, the *Context Recognising Service* handles the interaction with the corresponding *Context Observer* and removes the *Context* or adds it.

7. When a *Context* is requested to be added to or removed from a *Context Group*, the *Context Recognising Service* interacts with the corresponding *Context Group*. To identify the *Context* and *Context Group* unique names are used.

8. The *Context Group* handles the removal and addition of *Contexts* to it. This is done by firstly adding the *Context* to the *Context Group* and then passing the *Context Group* to the *Context* so it can send update notifications.

6.7.3 Context Recognition

The main part of the architecture is visualised and described in detail in this section. This workflow is started directly after the architecture has initialised itself.

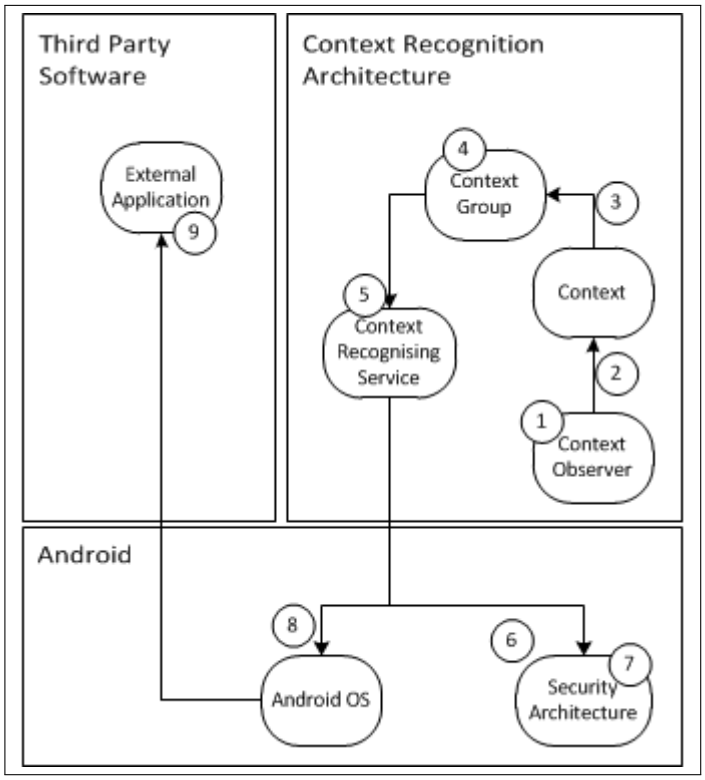

Figure 6.4: Workflow of the context recognition

1. The *Context Observers* start monitoring arbitrary information suitable to recognise the specific *Context*. The implemented *WifiObserver* scans the environment for available wireless networks every 5 seconds. Similar, the implemented *TimeObserver* takes the time every 5 seconds.

2. When a *Context* was recognised, the respective context receives a notification from the *Context Observer*. Respective, when a *Context* is left, a notification is sent as well.

3. The *Context* will then inform the *Context Group* about the state change. As a *Context* can be contained in an arbitrary amount of *Context Group*, every *Context Group* has to be notified.

4. Upon receiving a notification that one of the *Contexts* inside the *Context Group*, the state of every other *Context* contained in the *Context Group*. If all *Contexts* are active, the *Context Group* becomes active as well. Otherwise it will be inactive.

5. When the active state of a *Context Group* is changed, the corresponding *Context Group* notifies the *Context Recognising Service* about this. Depending on the state change of the *Context Group*, the *Context Recognising Service* handles this notification differently. If the *Context Group* becomes active, the *Context Recognising Service* pushes the *Context Group* at the top of an internal stack. Otherwise, the *Context Group* is removed from this stack.

6. After the *Context Recognising Service* receives an update from a *Context Group* and this update causes a different *Context Group* on top of the internal stack, the *Security Architecture* is notified with the ID bound to the now active *Context Group*.

7. The *Security Architecture* now switches the TAPL to the one identified by the id passed from the *Context Recognising Service*. After the switch is done, the *Security Architecture* checks whether the running processes are allowed to be executed in this now active *Context*. If an application is not allowed to be executed, it is terminated.

8. Additionally the *Context Recognition Service* sends a broadcast intent to the Android OS in case the *Context* has changed.

9. An external application can now receive this broadcast intent and react in its own way on this *Context* change.

7 Security Analysis

The implementation of a software that controls the execution of application on mobile devices needs to be as secure as possibly. When such a software is used to prevent employees from making photos of sensitive documents, the possibility arises that an employee is interested in breaking this mechanism. In this sample implementation, the service is the main target of attacks, as it controls everything and needs to expose itself so third party application can communicate with it. One possible attack would be retrieving vital data stored in the service and used by it in order to manipulate that data. Through IPC it is possible to get the data used by the service, however the original data isn't retrieved by this. By using the IPC implementation by the service, the data is copied to the calling process. However, since a management application needs as much control over the data contained in the service, such an application may need to add or even remove existing data to accommodate the service to new situations. The IPC implementation offers this functionality to third party applications. To secure this from misusing, every application that interacts with the service needs to ask for permission. The device administrator is informed about this permission in the moment he installs the application. Once this permission is granted, the application can interact with the service. The security of this IPC can be further enhanced by adding a password that needs to be entered before the other features become available. The Android system prevents third party applications from accessing the memory owned by the service. This is accomplished by separating applications by process, whereas each process has it own memory which can't be accessed by other applications that are executed in other processes. Another attack can be trying to manipulate the file that is used by the service to persist the context information on the local device. This file contains all the data that is used by the service during runtime and thus is a possible security risk. The Android system supports the protection of this file with mechanics of the Linux kernel. Every application is installed using a unique user account, created during installation for the application. This provides an application with it's own user directory in which it can store files to persist data. Files stored to this directory using the provided API of the Android system can only be accessed by the application to which the file belongs. This is done using the right management implemented in the Linux kernel, this

prevents third party applications from manipulating this vital file or deleting it. However, a possible attacker, as the user of the mobile device, has direct hardware access to this device by using a computer and the tools provided by Google Inc. to access the device and the data stored on it directly. This means, a possible attacker can connect the device to a computer and manipulate and access every file stored on the device. This opens a possible risk for security related applications installed on a mobile device. For instance, in the case of the sample implementation of the service, the attacker can connect an active mobile device to a computer und delete or manipulate the file containing the important data about the contexts used by the service. However, the service won't load any data from that file during runtime. This means that any changes made to that file during runtime won't take any effect on the service already running on the device. Moreover, the service will overwrite any changes made to that file once the service shuts down. As the service is constantly running as a background process, this usually only happens when the device shuts down as well and thus terminating the service as well. This means all changes made by an attacker will be overwritten by the service when the device is shutting down. When the attacker then starts the device again, right after the boot procedure is finished the service starts again and directly loads the data persisted in the file. After this is done, the service doesn't read from that file again unless the device was shutted down and started up again. The biggest security issue however is the removal of any part used by the service, including the service itself. While removing the management application or other third party application won't affect the service in any way, with having access to the device the attacker can also remove the service from it. However, even removing underlying structures, like kernel modules used by the service, will render the service useless.

8 Evaluation

This chapter discusses the assumptions made for this thesis as well as a conclusion and an outlook to future work relying on this research.

8.1 Development Assumptions

The security architecture proposed by this thesis was assumed to run on a mobile device utilizing the Linux kernel and being protected by an MTM. At the time of researching this, MTMs were specified, however not build into mobile devices yet, this was well-known while writing this thesis. However, TPM is widely built into laptops and PCs nowadays and implemented software, such as the Microsoft Bitlocker Drive Encryption, uses it to secure data on these device when lost or stolen. Thus, the assumption is made, that MTMs are going to be used in mobile devices soon as well.

Since the implementation was assumed to run on a Linux kernel, the Android OS was chosen to implement the security architecture. The assumption to run on a Linux kernel was made because is was necessary to modify the kernel. Due to its open-source nature, the Linux kernel, and thus the Android OS, met this requirement perfectly. Using the Android OS, there were mainly three ways to implement this security architecture. The first option was to use the abstraction provided by the OS to implement the context observation combined with low-level libraries to react on changes of contexts. The second option was to set the abstractions aside and implement the architecture completely as a low-level module loaded by the kernel at startup of the device. Lastly, the security architecture could have implemented directly inside the kernel as well.

8.1.1 Implementation

As already discussion in chapter 5, the implementation was done using the abstraction layer provided by the Android OS for observing the contexts and using low-level libraries to implement the TAPL switching. This offers the possibility to easily install this security architecture on a device as any other application as well. Furthermore, it can be extended,

for instance, new context observers can be added, and the update of can be spread to the device automatically. Implementating the architecture as a kernel module makes it harder to extend the architecture, as the abstraction of the Android OS are not available. In addition, the automatic update of application isn't possible here, which means the updated module has to be brought to each device and activated there manually. This is a downside when dealing with many devices. Also, it may be necessary to recompile the module if a device uses a different hardware, so that an employer needs to maintain multiple compilations of the same module, one for each target hardware. However, a huge drawback of both these implementation, whether using the abstraction or building a kernel module, is the possibility that an attacker can remove this from the device. When using the abstraction, the security architecture is installed like any other application on the device, but it can also be removed from the device like any other application. The same is valid for modules. Even if a module is currently in use by the kernel, it is possible to remove it by using the ADB. However, the device administrator can prevent applications that are able to uninstall other applications from running and thus making avoid the uninstallation of the implementation from the device. Furthermore, he can also disable the ADB for the device and preventing the preferences application from execution so that a possible attacker cannot change the settings made by the administrator. Implementing the security architecture directly into the kernel avoids having to deal with the removal of the security. However, the attacker can still flash the device to install a different kernel, which would also remove the architecture. Moreover, having a different kernel implementation becomes troublesome when the architecture needs to be updated or a new kernel updates are published, as the employer needs to maintain its own kernel.

8.1.2 Kernel Modifications

The data, including the file that contains the TAPL entries and binary data of the processes calling the exec() system-call which is used in computing the SHA256 hash value of the binary is read directly from the filesystem of the kernel space. Such tasks should be performed by a deamon providing the necessary abilities, in this case reading the data from the files. However, such a deamon must be started directly after the INIT process has finished and before any other process has started. Additionally, it has to be prevented that such a deamon can be replaced or bypassed during runtime. To eliminate these problems, the file is loaded within the kernel space. Note however, that this behaviour conflicts with the principle that kernel-space tasks may not be performed in user-space and vice versa.

8.1.3 Management Application

To test the implemented security architecture, Martin Landsmann developed an application that connects and allows the device administrator to interact with the security architecture. This application provides the capability of adding and removing applications from the TAPL. With the extension of the security architecture to provide context based execution control, the application was extended as well to support the new features. This software will be shown in this section.

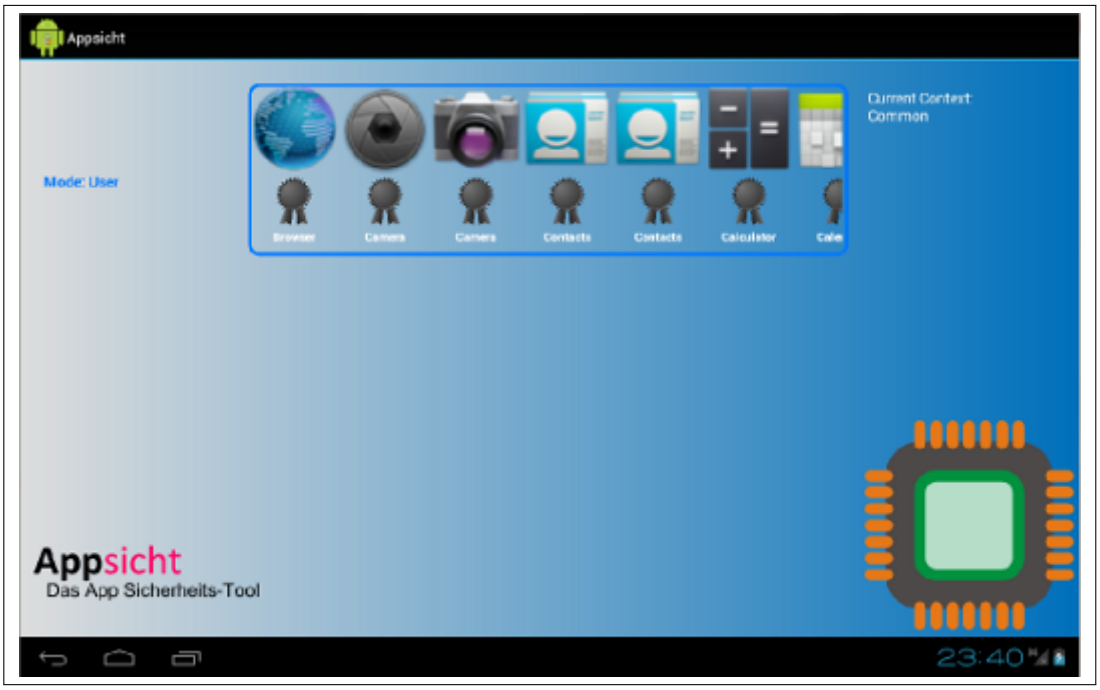

Figure 8.1: The startup screen of the application

Figure 8.1 shows the application as it appears right after the start. Shown are the information available for every user of the device, these are:

1. The mode the user is in, currently set to "User". A tap on this switches to administrator mode.

2. A list of applications installed on the device. Each entry consists of the icon of the application along with its name and in between an icon that visualises if the application is signed by the device administrator or not. When the user taps on the symbol, the application is started when it's signed.

3. An informational text about the current context the device is currently in.

At this point, the application has already connected to the context architecture running in the background and retrieves information from it, such as the current context the device is in. When the user switches to administrator mode, more options are added to the screen, as figure 8.2 shows.

Figure 8.2: The admin screen of the application

Most noticable, the screen colour has changed from a blue shade to an orange one. This helps the user to easily identify whether the application is in the user or administrator mode. Additionally, a dialog shows asking to enter the device administrator password for the context architecture. When entered, access to manipulating requests, like adding or deleting context, is granted. Otherwise these requests will be ignored.

1. The list of application is still shown as before, however, the behaviour has slightly changed. When tapping on an entry in this mode, the device administrator is asked if the application should be signed. When confirmed, the device administrator has to enter the passwort and the application will be signed. Otherwise, long tapping on an entry causes this application to be unsigned.

2. This button is only visible when the application is in administrator mode. It allows the device administrator to interact with the context architecture.

3. The second button is, as well as the other one, only visible when in administrator mode. It opens a screen for creating new and modify existing TAPLs.

The application list shown in this main view, whether the application is in administrator mode or not, always shows the applications which are signed in the current context, which is shown right of the list. All modification will be done to the TAPL that is bound to the active context. To modify a different TAPL, the administrator has to open the *TAPL View*.

When the device administrator taps on the button labeled with *Manage TAPLs* the *TAPL View* of the application is show:

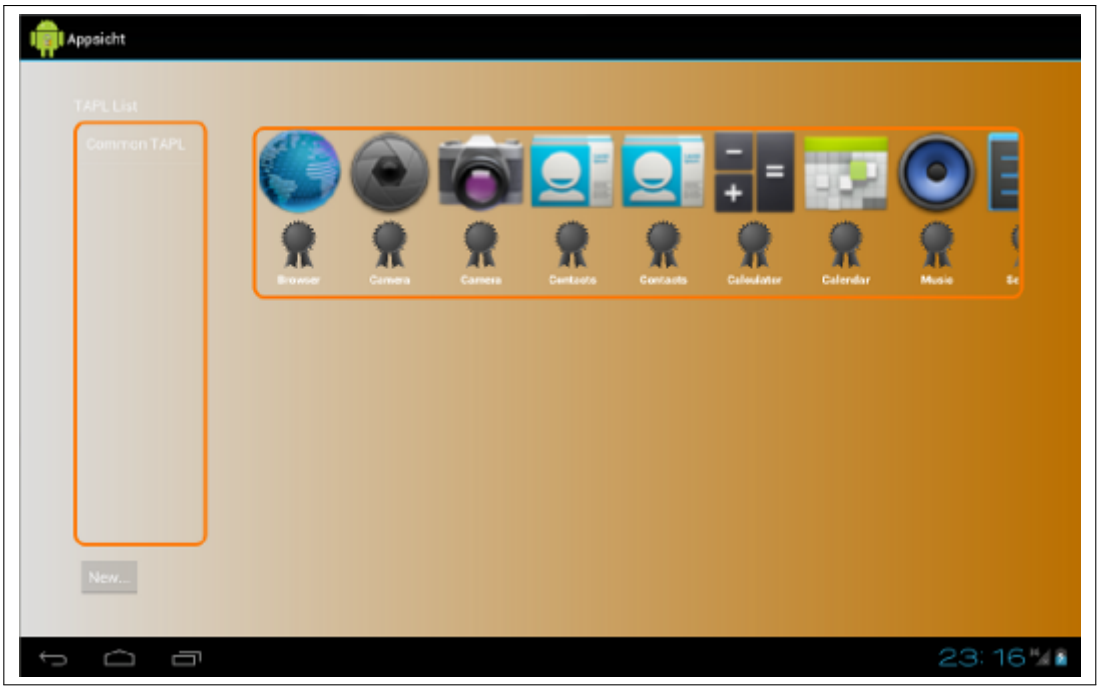

Figure 8.3: The *TAPL View*

This offers the device administrator to sign and unsign applications for a specific TAPL as well as creating new TAPLs. The elements shown in this view are:

1. A list of TAPLs known to the application, identified by name.

2. The list of installed applications on the device. Again, as in the *Main View* of the application, each entry consists of the icon and name of the application, as well as an icon to indicate if the application is signed or not.

3. A button to create a new TAPL in the security architecture.

When the device administrator taps on an entry in the TAPL list, the application list automatically update to show which applications are signed in the selected TAPL. By tapping on a application, the device administrator can now sign an unsigned application or unsign a singed application in the selected TAPL.

By tapping on the button labeled *New...* the device administrator is able to create a new TAPL. For this, a name must be provided to the dialog appearing shown in the following figure:

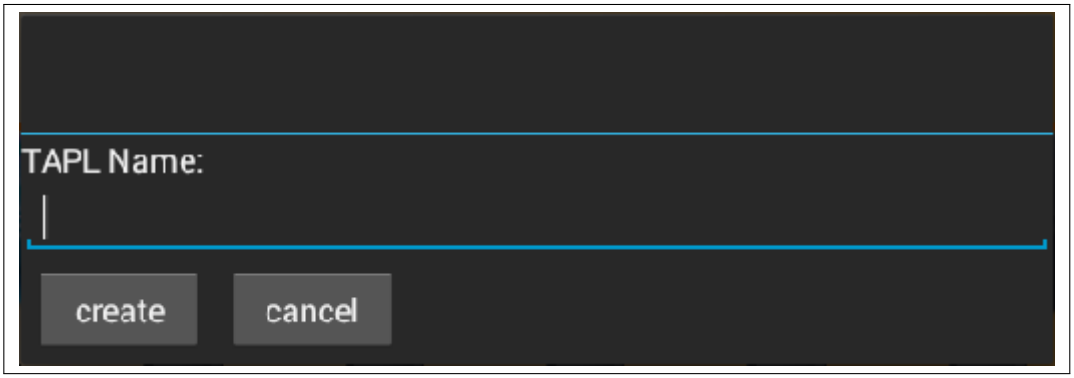

Figure 8.4: Creation of a new TAPL

The application then sends a request to the security architecture to create a new TAPL. As the security architecture identifies a TAPL with an unique integer id, this id is returned to the application. The application then binds the name entered by the administrator to the id returned by the security architecture. This helps to manage the TAPLs as it provides the possibility to give meaningful names to TAPLs instead of numbers.

When the device administrator taps on the button labeled *Manage Context* on the *Main View*, the *Context View* is shown. This view provides direct interaction with the context architecture running on the device. The elements of this view are described below as well as a visualisation of the view:

The *Context View* is parted into two halfs, the left one covers available interactions with context groups, whereas the right one interacts with the contexts inside the architecture.

1. This first list shows all available context groups inside the architecture.

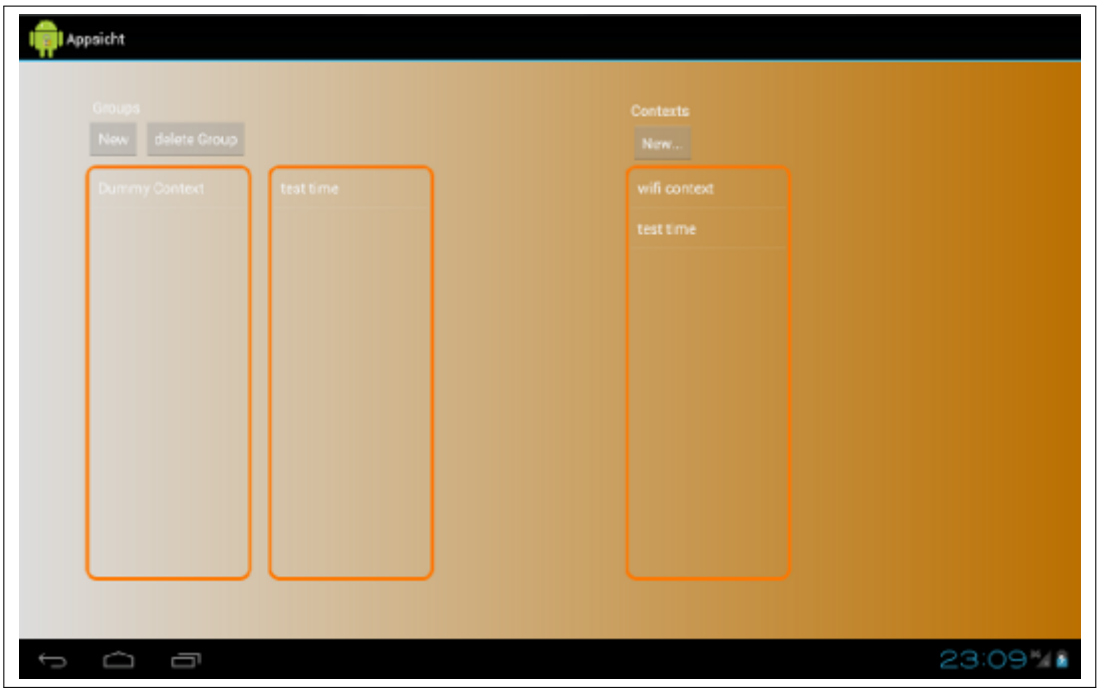

Figure 8.5: The *Context View*

2. The second list on the left half shows the context contained within a context group
3. The last list shows all context available in the context architecture.
4. This button enables the device administrator to create a new, empty context group and bind it to a existing TAPL.
5. When tapping this button, the device administrator deletes a previous from the list selected context group.
6. The last button on the right side allows the administrator to create a new context.

By tapping on the button labeled *New* on the left side of the screen, a dialog opens for creating a new dialog.

Here, the device administrator has to enter two names. The first one is the name of the group. As the context architecture identifies the groups by their names, this has to be unique. The second name is the name of the TAPL that will be bound to the group. Upon tapping on *create*, a request will be sent through the binder interface to the service running in the background with the provided data. To delete a group, the administrator simply taps on the

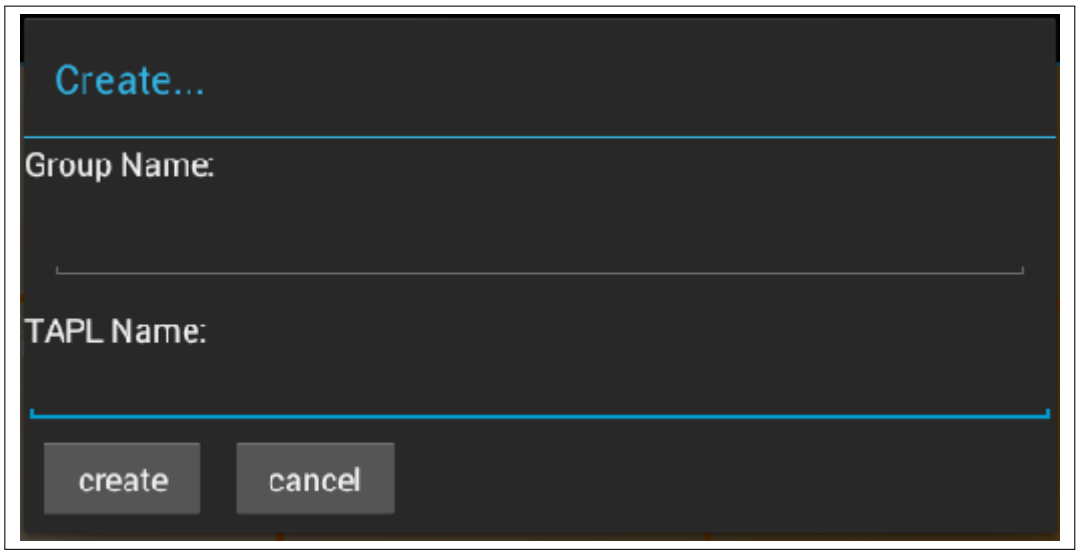

Figure 8.6: The new group dialog

group he wants to delete first and then a tap on the button labeled *delete Group* will send a request to the service to remove and delete that group.

The administrator can also use this *Context View* to add new contexts to the service running in the background. To do this, a tap on the button labeled *New...* on the right hand side of the screen opens a dialog based assistent. The first dialog that is shown is presented below:

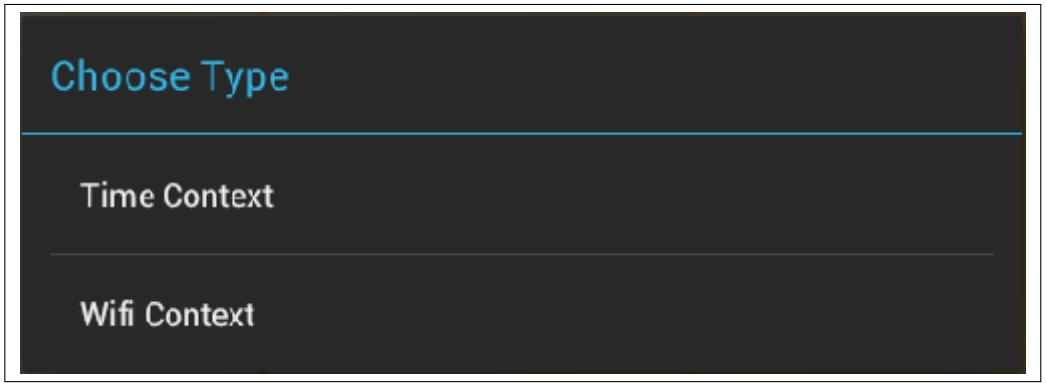

Figure 8.7: The creation of a new context

This first dialog presents the type of contexts available. The implementation currently only supports contexts based on wireless networks, *WifiContexts*, and contexts based on time, *TimeContexts*, but can be extended to support more kinds of contexts. The device administrator

can now select which type of context to create. After tapping on the corresponding entry, the next dialog shows and asks for the neccessary data:

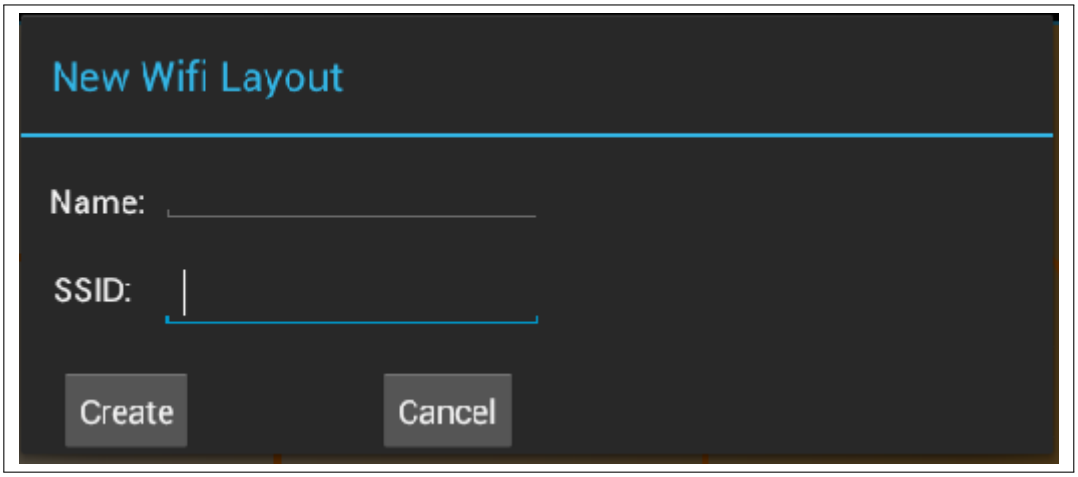

Figure 8.8: Creating a *WifiContext*

This illustrates the creation of a new *WifiContext*. The application asks the device administrator to enter the SSID the wireless network is identified by. Additionally, the context needs to have a name as well. As with the context groups, the service identifies the different contexts by their name, so these must be unique. After tapping on *Create* the application will create the context and sends it to the service. The context will then be copied and added.

The last step is to add the context to a context group. This is done by selecting a context group from the list on the left side and then long tapping on the context that is wished to be added. This will send a request to the service to add the context to the group.

8.2 Performance Evaluation

The performance overhead caused by the security architecture has been measured in (Landsmann, 2011). The measurements were taken in the OMAP4430 Panda Board. The OS and all application were loaded from an SDCard of Class 6. An ext3 type partition was used to install the OS. The kernel and bootloader were stored on a FAT32 type partition. Time measurements were performed using the kernel's high resolution timer defined in *linux/hrtimer.h* and were placed closely around the hooks from the security architecture. These measurements resulted in an overhead of ≈23.7s at startup. (Landsmann, 2011)

As the context recognition architecture is activated after startup, it does not affect the boot time in any way. The time needed to recognise a context depends on the procedure

implemented to recognise this context. The implementation in this thesis polls every 5s for new environmental information. This means, after the mobile device is in a defined context, it takes between 1 and 5 seconds for it to recognise this. Additionally, during the termination of unsigned application in the new context, no SHA256 hash has to be computed. According to (Landsmann, 2011), computation of SHA256 for Android applications are taking 0.58 $\frac{\mu s}{byte}$. The only computation needed to terminate unsigned application is iterating over the running application and comparing the names of the applications with the names in the currently active TAPL. In worst case, given *N* running applications and *M* applications in the TAPL, this results in comparing *N* times *M* string comparisions.

Another important factor is the power consumption of the mobile device. As the context recognition needs to be active at all times, the CPU can not suspend and save battery power. Moreover, depending on which kind of contexts are defined and how the observers are implemented, additional energy is used for recognition. The implementation presented in chapter 6 was tested on a Galaxy Nexus. The display of the mobile device was turned off, but GPS, WLAN and mobile connections were activ. The battery of the device has a capacity of $6.48Wh$ and was fully loaded at the beginning of the test. The active observers in the architecture were the *WifiObserver* and *TimeObserver* introduced in chapter 6. These observers are polling for new information every 5 seconds. In case of the *WifiObserver*, this means that every 5 seconds a scan for new networks is done. The *TimeObserver* gets the current time from the system every 5 seconds. The measurements were taken hourly and after each hour the capacity was reduced by 1% compared to the last measurement, meaning the capacity was reduced to 97% after 3 hours of testing.

$$6480 \text{ mWh total battery cap.} \times 3\% \text{ energy usage} = 194.4 \text{ mWh energy usage in 3 hours}$$

$$194.4 \text{ mWh energy usage in 3 hours} \div 3 \text{ hours of testing} = 64.8 \text{ mWh}$$

With the context recognition enabled and running in the background, the device consumes 64.8 mWh in stand-by. The power consumption was measured again on the same device but without the context recognition running. After 3 hours, the device used 2% of the battery capacity.

$$6480 \text{ mWh total battery cap.} \times 2\% \text{ energy usage} = 129.6 \text{ mWh energy usage in 3 hours}$$

$$129.6 \text{ mWh energy usage in 3 hours} \div 3 \text{ hours of testing} = 43.2 \text{ mWh}$$

$$64.8 \text{ mWh} - 43.2 \text{ mWh} = 21.6 \text{ mWh}$$

This shows, that the device uses roughly 21.6 mWh additional when the context recognition as implemented in chapter 6 is running.

8.3 Conclusion

The basic security architecture proposted by Martin Landsmann is capable to detect code manipulations after the installation of the code. To be able to tell if any code was manipulated, the SHA256 hash value from a binary is computed and signed to protect it against manipulation. When a binary was modified the SHA256 hash value differs from the one computed before und thus the manipulation is sucessfully detected. The manipulated binaries are then prevented from execution. Native Linux programs crucial to run the device are verified by the security architecture during the call to the exec() system-call. Android applications however, as they don't use the native Linux exec system-call, are verified using a hook inside the DVM. These two verification points cover all application starts on a mobile device using the Android OS and protect it against manipulation. This security architecture however is not able to verify any runtime loaded library or code that is already being executed by the device. It can detect any manipulation done to verified binaries without having knowledge about the type of manipulation and possible malicious behavior.

This thesis extends the basic security architecture by using environmental information to determine whether an application is allowed to run. To detect environmental changes, the security architecture uses a background process that observs the environment with through the hardware built inside the mobile device. A device administrator can use the environmental data to define several contexts which are then recognized by the device during runtime. Additional, multiple TAPLs can be created and linked with a defined context to control the execution of applications based on the context the device is in. Third party applications can be used to communicate with the background process to manage contexts and TAPL.

This security architecture proposes a whitelist approach to determine which application is allowed to be executed on the device. The whitelist is usually a subset of all the applications runable on the device.

8.4 Outlook

The context recognition can be secured against being stopped or interrupted by other applications or the device user.

The context recognition can be extended to allow remote administration with an application that receives data from a server and conntects to the recognition architecture.

The architecture can be extended by implementing additional context observer, e.g. making possible to have device users to check in via NFC before the mobile device can be used.

It may be possible to implement the context observation inside the kernel to reduce the additional power consumption.

Glossary

ADB (Android Debug Bridge), a versatile command line tool which enables the user to communicate with the Android emulator or with a device conntected to the PC (Google Inc., 2008a) 15, 36, 49

AIDL (Android Interface Defining Language), an IDL for the Android system that allows to define an interface for IPC (Google Inc., 2008b) 19

Android a Linux based OS for mobile devices created by Google Inc. (Wikipedia, 2012a) 1, 13–17, 22, 23, 26, 27, 32, 34, 35, 41, 45, 60, 61

API (Application Programming Interface), a specification intended to be used by software to communicate with each other or to provide services to other applications (Wikipedia, 2012b) 19, 46

augmented reality a live view of real-world environment whose elements are elements are augmented by computer-generated input (Wikipedia, 2012c) 1

AVM (Application Verification Module), a module used in the security architecture to verify an application 26, 30

black-list a list of entries indicating not accepted entries, contrary to a white-list 21, 22, 63

broadcast intent an intent broadcasted by Android or other applications on the mobile devices. that can be received by other applications 35

C programming language a low level, general-purpose programming language (Wikipedia, 2012e) 13, 14, 34, 61

C++ programming language a low level, general purpose and object oriented programming language (Wikipedia, 2012d) 13, 14, 19, 61

class an entity in object oriented programming 17

CPU (Central Processing Unit), the main part of a computer that computes informations and commands of applications (Elektronik-Kompendium.de, 2012a) 16, 17, 57

CRS (Context Recognising Service), the service located in the Java layer to recognise context changes 34, 35, 37, 38, 40

DVM (Dalvik Virtual Machine), a stack-based virtual machine for Java programming language used in the Android OS (Wikipedia, 2012f) 14, 16–19, 22, 23, 26, 27, 30, 58

emulator a hardware or software that emulates the functions of a different computer system (Wikipedia, 2012g) 60

forking creation of a new process using the fork() system call 18

garbage collector an algorithm that manages memory in programming languages such as the Java programming language. It automatically deletes memory that isn't used anymore 17, 62

Google Inc. an american IT company that provides Internet-related products (Wikipedia, 2012i) 23, 32, 47, 60

GPS (Global Positioning System), a space-based satellite navigation that provides location and time information (Wikipedia, 2012h) 1, 33, 57

heap memory with a dynamic size used by an application 17, 18

HMAC (Keyed-Hashing for Message Authentication), a mechanism for message authentication using cryptographic hash functions (Network Working Group, 1997) 25, 26, 30

IDL (Interface Defining Language), a specifig language to describe interfaces of software components (Wikipedia, 2012k) 60

inlining an optimisation where the call to a function is replaced by the code of the function that would have been called, thus saving the actual function call 18

intent a signal to Android to perform a specified action 60

IPC (Interprocess Communication), a method to exchange data between processes (Wikipedia, 2012j) 18, 19, 38, 46, 60

Java programming language an object oriented programming language(Wikipedia, 2012m) 13, 14, 17, 34, 61

JNI (Java Native Interface), a interface for the Java programming language that allows such programs to call to and to be called by native application and library written in languages such like the C programming language or the C++ programming language (Wikipedia, 2012l) 19, 20

kernel the main component of a OS, a bridge between applications and actual data processing at hardware level (Wikipedia, 2012n) 6, 13, 15, 17, 22–25, 27, 29, 30, 56, 59

kernel space memory area in which the kernel and its modules are operating, oposite to user-space 18, 19, 49, 63

Linux a modular, open source software OS (Wikipedia, 2012o) 13, 15, 22, 26, 27, 60

mark bit used by a garbage collector to mark objects that are in use 17

MLTM (Mobile Local-Owner Trusted Module), a MTM that provides a subset of the commands specified by the TPM version 1.2 and optionally additional commands (Trusted Computing Group, 2010) 3, 4, 7

MRTM (Mobile Remote-Owner Trusted Module), a MTM that provides a subset of the commands specified by the TPM version 1.2 and additionally mobile specific commands (Trusted Computing Group, 2010) 3, 4, 7, 21, 22, 24

MTM (Mobile Trusted Module), a module that provides security features to mobile devices, such as platform integrity and device authentication (Trusted Computing Group, 2010) 1, 3–7, 21, 22, 24–28, 48, 62

open source software computer software that is available in source code form (Wikipedia, 2012p) 13, 62

OS (operating system), a set of software that manages computer hardware and provides services for other software (Wikipedia, 2012q) 1, 6, 10, 11, 13–15, 22–25, 27, 32, 34, 41, 45, 56, 60–62

PAM (Process Authentification Module), a module used in the security architecture to authentificate an application 25, 26, 29, 30

PCR 3, 7

process an abstract instance of a computer program that is currently running (Wikipedia, 2012r) 14–19, 45

PVM (Process Verification Module) 25, 26, 29, 30

RAM (Random Access Memory), memory to store the data of running applications in (Elektronik-Kompendium.de, 2012b) 16, 17, 40, 63

RIM (Reference Integrity Metrics), is a reference value to compare a measurement against, for instance a SHA256 hash of a software image Trusted Computing Group (2010) 4, 7, 8, 21, 22, 24, 25, 27, 28

SHA256 A iterative, one-way hash function that can process a message to produce a condensed representation called a message digest (National Institute of Standards and Technology (NIST), 2002) 4, 27, 28, 30, 49, 57, 63

SMS (Short Message Service), a text messaging service of mobile communication devices (Wikipedia, 2012s) 2

SSID (Service Set Identifier), the name of a wireless network, according to the 802.11 standard 36, 39, 56

swap space memory that is used when the RAM is full 16

TAPL (Trusted Application List), a list that contains trusted applications that are allowed to run on the device 22, 24–26, 28, 31–33, 37, 39–41, 45, 48–50, 52–54, 57, 58

TCG (Trusted Computing Group), an international industry standard group that developes specification for trusted computing (Trusted Computing Group, 2012) 3

TPM (Trusted Platform Module), a module to provide basic security features 3, 4, 48, 62

user-space memory area in which user mode applications are working, contrary to kernel space (Wikipedia, 2012t) 18, 19, 62

virtual function table a table used by object oriented programming languages to determine which function has to be called in case of inheritance 17

white-list a list of entries indicating accepted entries, contrary to a black-list 21, 22, 60

Bibliography

[Elektronik-Kompendium.de 2012a] Elektronik-Kompendium.de: *CPU*. July 2012. – URL http://www.elektronik-kompendium.de/sites/com/0309161.htm. – visited: 2012/07/28

[Elektronik-Kompendium.de 2012b] Elektronik-Kompendium.de: *RAM*. July 2012. – URL http://www.elektronik-kompendium.de/sites/com/0309191.htm. – visited: 2012/07/28

[Google Inc. 2008a] Google Inc.: *Anatomy and Physiology of an Android*. June 2008. – Google I/O 2008

[Google Inc. 2008b] Google Inc.: *Dalvik Virtual Machine Internals*. June 2008. – Google I/O 2008

[Landsmann 2011] Landsmann, Martin: *Evaluating an MTM based security concept for Linux-kernel grounded mobile systems*. 2011

[National Institute of Standards and Technology (NIST) 2002] National Institute of Standards and Technology (NIST): *Secure Hash Standard*. August 2002

[Network Working Group 1997] Network Working Group: *RFC 2104: HMAC: Keyed-Hashing for Message Authentication*. February 1997

[Schreiber 2011] Schreiber, Thorsten: *Android Binder*. October 2011

[Trusted Computing Group 2007] Trusted Computing Group: *TPM Main Part 1 Design Principles*. July 2007. – Specification Version 1.2, Level 2 Revision 103

[Trusted Computing Group 2010] Trusted Computing Group: *Mobile Trusted Module Specifaction*. April 2010. – Revision 7.02

[Trusted Computing Group 2012] Trusted Computing Group: *About TCG*. July 2012. – URL http://www.trustedcomputinggroup.org/about_tcg. – visited: 2012/07/28

[Ugus und Westhoff 2011] Ugus, Osman ; Westhoff, Dirk: An MTM based Watchdog for Malware Famishment in Smartphones. In: Eichler, Gerald (Hrsg.) ; KÃ¼pper, Axel (Hrsg.) ; Schau, Volkmar (Hrsg.) ; Fouchal, HacÃ¨ne (Hrsg.) ; Unger, Herwig (Hrsg.): *IICS* Bd. P-186, GI, 2011, S. 251–262. – URL http://dblp.uni-trier.de/db/conf/iics/iics2011.html#UgusW11. – ISBN 978-3-88579-280-2

[Wikipedia 2012a] Wikipedia: *Android (operating system)*. July 2012. – URL http://en.wikipedia.org/wiki/Android_%28operating_system%29. – visited: 2012/07/28

[Wikipedia 2012b] Wikipedia: *Application programming interface*. July 2012. – URL http://en.wikipedia.org/wiki/Application_programming_interface. – visited: 2012/07/28

[Wikipedia 2012c] Wikipedia: *Augmented reality*. July 2012. – URL http://en.wikipedia.org/wiki/Augmented_reality. – visited: 2012/07/28

[Wikipedia 2012d] Wikipedia: *C++*. July 2012. – URL http://en.wikipedia.org/wiki/C%2B%2B. – visited: 2012/07/28

[Wikipedia 2012e] Wikipedia: *C (programming language*. July 2012. – URL http://en.wikipedia.org/wiki/C_%28programming_language%29. – visited: 2012/07/28

[Wikipedia 2012f] Wikipedia: *Dalvik (software)*. July 2012. – URL http://en.wikipedia.org/wiki/Dalvik_%28software%29. – visited: 2012/07/28

[Wikipedia 2012g] Wikipedia: *Emulator*. July 2012. – URL http://en.wikipedia.org/wiki/Emulator. – visited: 2012/07/28

[Wikipedia 2012h] Wikipedia: *Global Positioning System*. July 2012. – URL http://en.wikipedia.org/wiki/Global_Positioning_System. – visited: 2012/07/28

[Wikipedia 2012i] Wikipedia: *Google*. July 2012. – URL http://en.wikipedia.org/wiki/Google_Inc.. – visited: 2012/07/28

[Wikipedia 2012j] Wikipedia: *Inter-process communication*. July 2012. – URL http://en.wikipedia.org/wiki/Inter-process_communication. – visited: 2012/07/28

[Wikipedia 2012k] Wikipedia: *Interface description language*. July 2012. – URL http://en.wikipedia.org/wiki/Interface_description_language. – visited: 2012/07/28

[Wikipedia 2012l] Wikipedia: *Java Native Interface*. July 2012. – URL http://en.wikipedia.org/wiki/Java_Native_Interface. – visited: 2012/07/28

[Wikipedia 2012m] Wikipedia: *Java (programming language)*. July 2012. – URL http://en.wikipedia.org/wiki/Java_%28programming_language%29. – visited: 2012/07/28

[Wikipedia 2012n] Wikipedia: *Kerne (computing)*. July 2012. – URL http://en.wikipedia.org/wiki/Kernel_%28computing%29l. – visited: 2012/07/28

[Wikipedia 2012o] Wikipedia: *Linux*. July 2012. – URL http://en.wikipedia.org/wiki/Linux. – visited: 2012/07/28

[Wikipedia 2012p] Wikipedia: *Open-source software*. July 2012. – URL http://en.wikipedia.org/wiki/Open-source_software. – visited: 2012/07/28

[Wikipedia 2012q] Wikipedia: *Operating System*. July 2012. – URL http://en.wikipedia.org/wiki/Operating_system. – visited: 2012/07/28

[Wikipedia 2012r] Wikipedia: *Process (computing)*. July 2012. – URL http://en.wikipedia.org/wiki/Process_%28computing%29. – visited: 2012/07/28

[Wikipedia 2012s] Wikipedia: *Short Message Service*. July 2012. – URL http://en.wikipedia.org/wiki/SMS. – visited: 2012/07/28

[Wikipedia 2012t] Wikipedia: *User space*. July 2012. – URL http://en.wikipedia.org/wiki/User_space. – visited: 2012/07/28

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 15. August 2012 Johannes Wilken