

# Bachelor Thesis

Martin Landsmann

Evaluating an MTM based security concept for Linux-kernel  
grounded mobile systems

Martin Landsmann

**Evaluating an MTM based security concept for Linux-kernel  
grounded mobile systems**

Bachelor Thesis eingereicht im Rahmen der Bachelor

im Studiengang Bachelor of Science Technische Informatik  
am Department Informatik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Dirk Westhoff, Prof. Dr.-Ing. Martin Hübner

Eingereicht am: 14. December 2011

**Martin Landsmann**

**Thema der Arbeit**

Evaluating an MTM based security concept for Linux-kernel grounded mobile systems

**Stichworte**

IT-Sicherheit, Mobile Sicherheit, Remote Attestation, MTM, Kernel, Linux, Android OS

**Kurzzusammenfassung**

Intelligente mobile Geräte, wie Smartphones, werden zunehmend in Unternehmen genutzt, um mobilität in den produktiven Prozess der Arbeit zu integrieren. Korrespondenz, sowie Übertragung von Dokumenten, also auch eine Verbindung zu Unternehmens-internen Daten wird durch den Einsatz solcher Geräte ermöglicht. Mit diesen Möglichkeiten sind sie auch ein lohnendes Ziel um vertrauliche Informationen sowie Daten zu entwenden oder diese zu manipulieren. Dieses Ausarbeitung beschreibt die Implementierung sowie die Evaluation eines Sicherheitskonzepts für Linux Kernel basierte mobile Geräte. Der Schwerpunkt liegt darin, Manipulationen an ausführbarem Code zu erkennen und zu verhindern, daß manipulierter Code zu Laufzeit ausgeführt werden kann. Dieses Ziel wird mithilfe einer listen-basierten Überprüfung des ausführbaren Codes erfolgen. Damit wird verhindert, daß manipulierter Code Schaden an dem Gerät sowie an anderen, möglicherweise vertraulichen, Daten verursachen kann.

**Martin Landsmann**

**Title of the paper**

Evaluating an MTM based security concept for Linux-kernel grounded mobile systems

**Keywords**

IT-Security, Mobile Security, Remote Attestation, MTM, Kernel, Linux, Android OS

**Abstract**

Sophisticated mobile devices like smartphones are increasingly used in companies to integrate mobility into the productive process of work. Correspondence, transmission of documents, as well as the connection to Company-internal data is possible by using such devices. With these options, they are also a worthwhile target to steal confidential information and data, or manipulate it. This thesis describes the implementation and evaluation of a security concept for Linux kernel-based mobile devices. The focus is to detect manipulations performed on executable code and to prevent such manipulated code to be executed at runtime. This will be assured using a list-based verification of the executable code. Such verification and execution control prevents damage to the mobile device and to potentially sensitive data.

# Contents

<b>1</b>	<b>Problem Statement</b>	<b>1</b>
<b>2</b>	<b>Main Concept of the Security Architecture</b>	<b>3</b>
2.1	Security Goal . . . . .	3
2.2	Secure Startup . . . . .	4
2.3	Program Certification . . . . .	4
2.4	Program Verification . . . . .	5
2.5	Program Execution Control and Verification at Runtime . . . . .	5
<b>3</b>	<b>Mobile Trusted Module - MTM</b>	<b>6</b>
3.1	Secure Boot . . . . .	7
3.2	Reference Integrity Metric Certificate . . . . .	9
3.2.1	Counters . . . . .	9
3.2.2	Keys . . . . .	10
3.3	MRTM - Mobile Remote-owner Trusted Module . . . . .	11
3.4	MLTM - Mobile Local-Owner Trusted Module . . . . .	11
<b>4</b>	<b>Linux Kernel</b>	<b>13</b>
4.1	Device Nodes . . . . .	14
4.2	Kernel Modules . . . . .	15
4.3	Process . . . . .	16
4.3.1	Process Creation . . . . .	16
4.4	Task . . . . .	17
4.4.1	Process Identification Number (PID) . . . . .	17
4.4.2	Process State . . . . .	18
4.5	Thread Scheduling . . . . .	18
<b>5</b>	<b>Android Operating System</b>	<b>20</b>
5.1	Android OS Kernel . . . . .	20
5.2	Android OS Library Layer . . . . .	21
5.3	Android OS Runtime . . . . .	22
5.3.1	Dalvik VM . . . . .	22
5.4	Android Application Framework . . . . .	22
5.5	Android Applications . . . . .	23
5.6	Android Boot Sequence . . . . .	23
5.6.1	Android Application Start . . . . .	25

<b>6</b>	<b>Security Architecture</b>	<b>26</b>
6.1	Architecture Modules . . . . .	28
6.1.1	Security Architecture Keys . . . . .	28
6.1.2	MRTM Wrapper (MTM) . . . . .	28
6.1.3	Trusted Application List (TAPL) . . . . .	29
6.1.4	Application Verification Module (AVM) . . . . .	29
6.1.5	Application Signing User Interface (AppSigningUI) . . . . .	29
6.1.6	Process Authentication Module (PAM) . . . . .	29
6.1.7	Process Verification Module (PVM) . . . . .	29
6.2	System Integration . . . . .	30
6.2.1	Process Creation . . . . .	30
6.2.2	Boot Sequence . . . . .	30
6.2.3	Starting a new Program (Linux) . . . . .	30
6.2.4	Starting a new Application (Android OS) . . . . .	31
6.2.5	Administration . . . . .	31
6.3	Security Analysis and Discussion . . . . .	32
<b>7</b>	<b>Implementation</b>	<b>34</b>
7.1	Kernel Modifications . . . . .	34
7.1.1	fork.c Modifications . . . . .	35
7.1.2	sys_arm.c Modifications . . . . .	37
7.2	Security Architecture Modules . . . . .	37
7.2.1	sec_tapl.ko Module (sec_TAPL) . . . . .	37
7.2.2	sec_autoload.ko Module . . . . .	38
7.2.3	sec_cryptoavm.ko Module (sec_AVM) . . . . .	38
7.2.4	sec_key_storage.ko Module (sec_Key_Storage) . . . . .	38
7.2.5	sec_rsacryptomtm.ko Module (sec_MTM) . . . . .	38
7.2.6	sec_ui.ko Module (sec_UI) . . . . .	39
7.2.7	sec_verify_bridge.ko Module (sec_Verify_Bridge) . . . . .	40
7.3	Android DVM Core Modifications . . . . .	40
7.4	Implementation Workflow . . . . .	41
7.4.1	Workflow for the Linux Kernel . . . . .	41
7.4.2	Workflow for Android OS Layer . . . . .	43
<b>8</b>	<b>Evaluation</b>	<b>48</b>
8.1	Development Assumptions . . . . .	48
8.1.1	General Assumptions . . . . .	48
8.1.2	Kernel Modifications . . . . .	49
8.1.3	PAM and PVM . . . . .	49
8.1.4	Cryptographic Abilities . . . . .	50
8.2	Development Platform and Prototyping . . . . .	50
8.2.1	Command Line Configuration Tool - sec_configure Tool . . . . .	50
8.2.2	GUI for the configuration Tool - sec_Configuration Tool . . . . .	53

*Contents*

---

8.2.3	View - TAPL . . . . .	56
8.2.4	Automated Build Script . . . . .	57
8.3	Performance Evaluation . . . . .	58
8.4	Conclusion . . . . .	61
8.5	Outlook . . . . .	62
	<b>Bibliography</b>	<b>64</b>
	<b>Glossary</b>	<b>68</b>

# 1 Problem Statement

Mobile devices like smart-phones and tablet PCs<sup>1</sup> are permanently advancing. Due to their enormous popularity, the number of smart-phone users are increasing constantly. The number of programs available for smart-phones rises, coming often from an unknown origin, e.g. from third-party developers. The growing abilities of these devices make them more sophisticated and usable in a wide range of applications. They can be integrated into the home and business network topologies, for extending mobility and accessibility of users. These benefits rise also the question about the security and secrecy vulnerabilities introduced by these devices. Protecting sensitive and private data against unauthorised access is a major security requirement for mobile devices. It is essential to fully integrate a device into a network, dealing with vulnerable data and services. The devices have to be secured against threats of an environment before integrating them. An insufficient protection can be exploited by attackers to corrupt the security of the whole topology. Such a vulnerability can be used to bypass security mechanisms, to grant access to sensible data and services, or even to cause serious damage to the device and other connected units. A protection against system manipulation and other attacks has to be ensured for mobile devices, mainly because they are capable of running third-party programs from untrusted origins. Such programs have high exposure, as they can have malicious behaviour. Desktop PCs and Laptops<sup>2</sup> can be protected with anti-virus and cognate programs against attacks. Such a protection mechanism is not feasible for mobile devices. Their hardware capabilities and as a result of their compact design, their energy supplies are limited. A comprehensive scan of system files against attacks would absorb a major amount of their computation and energy resources. Thus, such scans would significantly lower their power-on time. However, no or a less comprehensive protection is not an option in this scope. As such applied devices without proper security mechanisms applied are never accepted to be integrated in networks providing access to confident data e.g., company networks. Any kind of installed and executed programs can harm the device and the interacting environment, as malicious behaviour of an installed program is not always obvious.

User-sided careless handling or ignorance of possible threat endangers the whole topologies'

---

<sup>1</sup>PCs (Personal Computer) is a small computer for individual personal use [WIKI].

<sup>2</sup>Laptops is a mobile PC with independent power supply.

security. Different approaches have been established to protect mobile platforms to protect devices against various vulnerabilities. Symbian<sup>3</sup>'s trust policy distinguishes between trusted and untrusted applications and subsequently prevents them from interacting with each other. Data is as well protected against untrusted applications. This is done by marking data as readable and editable from trusted and untrusted programs. Symbian distinguishes what type of program tries to use or edit the marked data and controls the access to it. Apple<sup>4</sup>'s market-place applications are exclusively allowed to be executed on an Iphone/Ipad and no third-party programs are installable or executable on them. However, signing programs for the mobile devices is out of a local administrator's control. This situation is unsatisfying as the administrator has no knowledge about which criteria have been applied to allow the program being signed. To restore the administrator's sovereignty over the mobile device, a security architecture is required which allows the administrator exclusively to sign programs. Moreover, this mitigates the necessity to scan the device permanently for possible malicious code, because programs are verified before they are executed and uncertified programs are prevented from execution. Only if a program contains a valid signature, providing that the program has not been manipulated, it is allowed to be executed. A local device administrator is allowed to certify programs and subsequently stores the certificates on the device. Such an approach grants the administrator full control which programs may run on the administrated device. In addition, programs can be tested in a closed environment before certified as trustworthy and can chosen to be trustworthy or not trustworthy. Such a distinct classification of programs limits the ground for attacks, and furthermore it reduces the administrative expense. During power-on time, manipulated programs are prevented from execution, as they are not matching their certificate any-more. This makes the device intangible for malicious programs and attacks based on malicious code manipulating other programs, even for yet previously unknown malicious code. The proposed approach assumes secrecy at the certification process and supposes the possibility to detect manipulations while the device has been deactivated. The certification check has to be atomic. Moreover, it has to defy attacks against it. In this work, all the pointed properties are merged into a security concept for mobile devices. The security concept can be indicated through the following four basis characteristics:

- Activating the device introduces a secured state. (Secure Boot)
- A not conquerable atomic certificate verification process is provided. (Program Verification)
- Only certified and verified programs are allowed to execute. (Verified Program Execution)
- Certifying programs is only allowed from within a secured state. (Program Certification)

---

<sup>3</sup>Symbian is an operating system for mobile devices [Wikj].

<sup>4</sup>Apple is a company providing operating systems, mobile devices and home and business computer [Www].



## 2 Main Concept of the Security Architecture

This chapter introduces the security architecture applying a hardware based verification to proof the device integrity. It points out the coverage and the preconditions for a firm security architecture. Similar concepts are partly discussed in [Mtm], [Nau+10] and [UW11]. The requirements and the boundaries of the introduced security architecture are discussed as well. The security architecture consists of four major parts or stages establishing a working and firm architecture. These are:

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

### 2.1 Security Goal

The proposed security architecture provides a secure startup (cf. Section 2.2) boot<sup>1</sup>ing a mobile device in a trustworthy state. This should prevent uncertified code from execution. Altered programs can be detected by this security architecture and are subsequently prevented from executing. Moreover, this security architecture provides certificate creation and handling directly on the device, making the certificates uniquely usable with it.

Only legitimated users (administrators) are allowed to request certification of programs and such architecture should be orthogonal to any existing operating system<sup>2</sup> policies, restrictions and should not be possible to be bypassed. It should be hardware based and provide a secure key storage to secure the keys against extraction or manipulation. However, this security architecture does not cover the protection against all possible threats. It is designed to aid the present system's security mechanisms to lower the computational overhead and with it the power consumption compared to other solutions like e.g. anti-malware programs.

---

<sup>1</sup>**boot** describes the initial operations a computer performs when switched on [Wika].

<sup>2</sup>**operating system** "An operating system is a set of programs that manage computer hardware resources and provide common services for application software" [Wikf].

## 2.2 Secure Startup

The secure startup implements a monitored startup process beginning from power-on that boots the mobile device into a well known and trusted state. The secure startup is a basic requirement for the proposed security architecture. Activating a deactivated mobile device forces security checks that provide a trust anchor<sup>3</sup> for other security related components of the architecture. These security checks include the following steps. Firstly, the hardware has to be checked for possible manipulations. Secondly, the devices' flashed image<sup>4</sup> has to be checked and thirdly the bootloader<sup>5</sup> for the operating system needs to be checked. The checks are performed by comparing calculated metrics. At startup, metrics are taken and calculated from the flashed image, the bootloader and the operating system. They are compared to metrics taken and calculated earlier in a secured and trustworthy state. These trustworthy metrics are protected against manipulation or exchange. They are stored in immutable storage locations and only accessible if the system is in a secured state. Additionally, the calculated metrics are encrypted to protect them against recreating equivalent entries matching manipulated or untrusted programs. These checks and functions require a hardware module to be present, providing measuring, cryptographic and storing abilities. Such hardware separates the initial trust anchor of the secured startup completely from the running architecture on the mobile device, as its purpose does not rely on the running architecture. Even so, the bootloader and the operating system must own and ensure to provide this trust anchor. After a successful secured startup, the mobile device is in a traceable trustworthy state. This approach protects a stolen device against undetected manipulation. An attacker could steal such a device and would have sufficient amount of time to attempt manipulations which would be revealed at the secure startup.

## 2.3 Program Certification

The certification process is exclusively allowed to the administrator of a mobile device. This grants the administrator full control of which programs are allowed to be executed. A certificate creation is performed by the mobile device. Only an authorised administrator can request this process. This requires an authentication and verification<sup>6</sup> interface from the mounted cryptographic hardware to authenticate and grant a certification request only to them. To create a certificate,

---

<sup>3</sup>**trust anchor** describes a reliable well known and trusted state from which a latter state can extend.

<sup>4</sup>**flashed image** is a software component stored into a permanent memory [Wikc].

<sup>5</sup>**bootloader** manages the boot process [Wika].

<sup>6</sup>**verification** is a mathematical operation which verifies the origin of a message/data [MOV01, ch.11.1].

the administrator passes a key representing the program, e.g. a hash<sup>7</sup>, and other information about the program to the cryptographic hardware. Then, the cryptographic hardware creates a certificate using these information and signs<sup>8</sup> it. The created certificates are signed using a private key only known to the hardware module. The private key is sealed inside it and never released. The public key is as well sealed inside it. In contrary to the private key, this public key is released to the system after the secure startup (cf. Section 2.2) to be used at the runtime for program verifications (cf. Section 2.4). The signed certificates are used to compare the provided hash of the program with an actual computed hash value of it. This reveals manipulation on them if the hash values does not match.

### 2.4 Program Verification

To verify a program, its hash value is computed. The corresponding certificate is then looked up from a list of all existing certificates on the mobile device. As the certificates are signed with the private key known only to the hardware module, the list can be placed on an untrusted storage. Manipulating the signed certificates directly would invalidate them. Assuming the cryptographic algorithm used for signing as cryptographically strong, it is not feasible to manipulate or recreate the certificates successfully. The calculated hash value is then verified using the certificate and the public key.

### 2.5 Program Execution Control and Verification at Runtime

To keep the trust chain valid after the secure startup (cf. Section 2.2), programs have to be verified at runtime<sup>9</sup> e.g. before they are executed. Every program execution is handled by the operating system. In this architecture the operating system verifies the program just before it is executed, as described in Section 2.4. After a successful verification, the execution will be continued by the operating system. In case of an unsuccessful verification, the execution is aborted immediately. This prevents maliciously manipulated programs from execution and from a possible system manipulation [UW11].

---

<sup>7</sup> **hash** is a function that calculates a key value from a set of data. This key value is typically smaller in size than the size of the data. It can be used to accelerate sorting and searching operations [KKP09, Ch.5], [MOV01, ch.9.2.1].

<sup>8</sup> **signs** is a mathematical operation which assures the origin of a message/data [MOV01, ch.11.1].

<sup>9</sup> **runtime** is the time after the boot process until the power off. During this period of time non operating system software can be used.

### 3 Mobile Trusted Module - MTM

This chapter discusses the Mobile Trusted Module (MTM<sup>1</sup>). Its main concept is introduced here. Related to the proposed security architecture, operation modes, key handling and the capabilities of this hardware are pointed out. Protection mechanisms and approaches for the MTM parts are also discussed.

The MTM has been specified by the Trusted Computing Group (TCG<sup>2</sup>). It is a tailored version of the Trusted Platform Module (TPM<sup>3</sup>) proposed for desktop PCs [Tpm]. The specification of the MTM describes the MTM as an aggregation of processing functions and services, with the ability to attest their trust and report its current state [Mtm, p.13-14]. It is a hardware module which provides integrity measurements for mobile devices. It can take metrics from hardware and software components, store them and compare them with reference values. All metric results are stored inside the MTM, separating them completely from other device components. This makes the stored results intangible to them and protects the metrics against direct access and manipulation. The MTM is able to measure and ensure its own integrity. It can verify the mobile device as trustworthy by attesting its integrity<sup>4</sup>. This provides a remote attestation<sup>5</sup>, as a remote entity can trust the measurements and results from the MTM. A remote attestation is required to protect unauthorised access to confidential data and services of the mobile device, e.g. online banking data. The MTM provides trusted services to the mobile device. These services are capable of performing integrity attestations on calling services. This capability is used and accessed by the device itself. Services such as an online banking software can proof the device as to be not manipulated to the banking server which assures e.g. a safe and accurate communication. The integrity measurements are stored inside the MTM. The storage for these values is a shielded location where the data can be stored and manipulated safely by the MTM without losing its integrity. These shielded storage locations are called Platform Configuration Registers (PCRs<sup>6</sup>).

---

<sup>1</sup>MTM (Mobile Trusted Module) is a hardware-based integrity assurance for mobile devices [Mtm].

<sup>2</sup>TCG (Trusted Computing Group) is an organisation for standards.

<sup>3</sup>TPM (Trusted Platform Module) is an integrity measurement and verification module for desktop PCs and Laptops [Tpm].

<sup>4</sup>integrity is a metric that indicates a well known state asumed as healthy.

<sup>5</sup>remote attestation Remote attestation describes a process where a service provider can attest an entity calling a service as not manipulated without physically access this entity.

<sup>6</sup>PCRs (Platform Configuration Registers) are immutable storing registers only accessible by the MTM.

The access to the PCRs is exclusively granted and handled by the MTM [Tpm, p.27-28]. The MTM specification elaborates two core domains for the MTM. They differ in the utilisation and the accessibility to the mobile device. These domains are fitted for two stakeholders. That is, the local-owner, usually represented by the user of the mobile device, and the remote-owner, usually represented by the device's manufacturer and service provider. The local-owner has physical access to the device and uses its services directly, e.g. the user synchronises the contact data from the device with the contact data on his home server. The remote-owner provides accessible remote services and/or data to the mobile device, e.g. transact money through an online banking service. The MTM is configured at manufacturing time with the capabilities and functions for either stakeholders' scope. With a set configuration matching one of the stakeholders, the MTM is differentiated either into the Mobile Local-owner Trusted Module (MLTM<sup>7</sup>) for the local-owner, or the Mobile Remote-owner Trusted Module (MRTM<sup>8</sup>) for the remote-owner. Due to their different scopes of application, the MLTM and MRTM have different approaches to handle attestations and verifications. The MRTM must protect the trust of the device from the power-on (cf. Section 3.1), to assure that no manipulation occurred before remote access. This point is the major difference between MRTM and MLTM. The MLTM can provide a secure boot [Mtm].

## 3.1 Secure Boot

To establish the trust on the device starting from power-on, and ending in a trustworthy state, the integrity of the mobile device is measured and compared to reference values stored in the MTM. If these checks fail, the secure boot terminates the booting process. Secure boot provides a transitive chain of trust, passing the trust of one attested layer to the next layer above until the operating system is booted. The secure boot sequence is illustrated in Figure 3.1 and composed of the following steps:

- ① At runtime a piece of code, stored inside the MTM, is executed measuring the MTM's own integrity and verifying it
- ② The involved modules are called Root-of-Trust-for-Verification (RTV<sup>9</sup>) and Root-of-Trust-for-Measurement (RTM<sup>10</sup>). The RTM calculates metrics of itself and the RTV and these

---

<sup>7</sup>MLTM (Mobile Local-owner Trusted Module) is a MTM with local ownership setup.

<sup>8</sup>MRTM Mobile Remote-owner Trusted Module is a MTM with remote ownership setup.

<sup>9</sup>RTV (Root-of-Trust-for-Verification) is a verification module inside the MTM.

<sup>10</sup>RTM (Root-of-Trust-Measurement) is a measurement module inside the MTM.

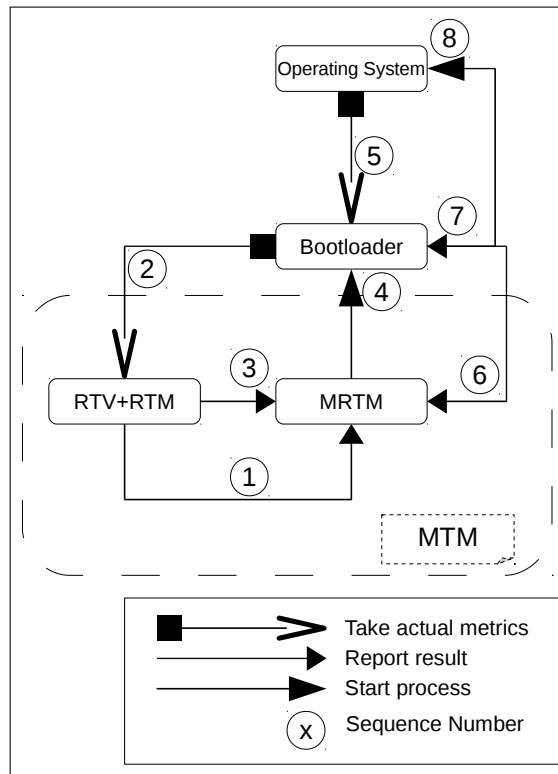


Figure 3.1: MTM secure boot sequence (cf. [Mtm, p.15] Figure 2 - Overview of MRTM)

metrics are subsequently verified by the RTV. After successful verification, the MTM is measured and verified by RTM and RTV. Finally RTV and RTM measures the bootloader.

- ③ This check is done by computing and comparing the hash of the actual bootloader's state with a computed reference hash taken earlier in a secure state, e.g. at manufacture time. The reference hash represents a trustworthy state of the bootloader. This hash verifies the bootloader to be trustworthy, if it equals the actually taken hash.
- ④ After the bootloader has been verified, the further measuring task is delegated to the bootloader.
- ⑤ The bootloader measures the integrity of the operating system's kernel<sup>11</sup>
- ⑥ and passes the results to the MTM.
- ⑦ The MTM again compares the computed hash with a reference hash computed earlier and passes the result back to the bootloader.

<sup>11</sup>kernel is the core software component of the Linux operating system [Lov05, p.32].

- ⑧ Now, the bootloader starts the verified kernel.

The kernel then performs the next measurements. It measures the integrity of modules which have to be loaded during the boot process. Finally, the operating system's components are verified by the kernel. A failure at any of the above pointed steps results in the termination of the boot process. This chain leads to a trustworthy state which verifies a trustworthy and not manipulated device whose integrity is ensured up to the operating system level [Mtm, p.97].

## 3.2 Reference Integrity Metric Certificate

A Reference Integrity Metric (RIM<sup>12</sup>) provides reference metrics and other additional information for its corresponding entity, e.g. a program binary file. They are used to verify the integrity of the measured entity, e.g. the bootloader or the device hardware, according to already taken metrics. The RIMs are stored in PCRs to protect them against manipulation. A RIM itself is certified to ensure that it is derived from a trusted origin and to protect it against manipulation before it is stored within a PCR. A RIM-certificate can be a signed hash computed of the RIM, plus some additional information vouching the origin and the trust of this certificate, e.g. a shared secret. RIM-certificates can be generated inside or outside the MTM. The internally generated RIM-certificates are automatically assured as a trusted certificate. This can be done because the MTM supervises and protects the creation of RIM-certificates. These internal RIM-certificates are bound to their generating MTM and cannot be used in other MTMs (other devices). Externally generated RIM-certificates can be used by a variety of platforms. They are certified by internal RIM-certificates, keys or other authenticating methods before they are accepted. They can be converted into internal RIMs by the MTM if necessary. Each MTM can authorise and authenticate its RIM-certificates. Only authorised parties (RIM\_auth parties) can create authentic RIM-certificates. They can be either the MTM or a foreign entity. RIM-certificates are authenticated using digital signatures or keyed message authentication codes (HMACs<sup>13</sup>) [Mtm, p.18-19].

### 3.2.1 Counters

#### **counterRIMprotect**

To protect the internal RIM-certificates against re-flashing and similar attacks, the MTM provides the counterRIMprotect counter. A RIM-certificate provides a “counter-stamp” field. This field is

---

<sup>12</sup>RIM (Reference Integrity Metric) is a calculated metric of an entity with further information about this entity [Mtm, p. 18].

<sup>13</sup>HMACs (Hash based Message Authentication Code) [MOV01, ch.9.5, 9.67].

compared to the actual counterRIMprotect counter value vouching the RIM-certificates freshness. The counterRIMprotect is a monotonic counter. It can only be increased and has a proposed limit of 4095. Increasing this counter causes all internal RIM-certificates having a lower value to become outdated. This causes the corresponding RIMs to be untrusted [Mtm, p.34].

#### **CounterBootstrap**

The device is protected against flash<sup>14</sup>ing a new firmware<sup>15</sup> by another counter. This Counter-Bootstrap counter is also increased monotonously, and out-dates the RIM-certificates for the “old” firmware image. The “new” firmware image comes along with an external RIM-certificates validating it. The CounterBootstrap counter limit is proposed to be 31. Both counters are placed in PCRs to be shielded against manipulation. They are not increased, until all operations affecting counter-values or “counter-stamps” were performed successfully [Mtm, p.35, 50].

#### **3.2.2 Keys**

##### **Endorsement Key**

The Endorsement Key (EK<sup>16</sup>) is unique to each MTM. It is used to sign RIM-certificates for its MTM and cannot be used for other MTMs. The EK is the private key part of the private-public-key pair. This key is only known to authorised parties that are allowed to create RIM-certificates. Depending on the configuration of the MTM, if it is set to MLTM or MRTM, the EK can be stored in a PCR, outside the MTM, or both [Mtm, p.8, 29, 82], [Tpm, p.29, 55].

##### **Verification Key**

The verification key is used by the MTM to verify RIM-certificates. Typically it is the corresponding public-key part of the EK. Verification keys can be stored and handled in a key hierarchy, where the root key is called Root Verification Authority Identifier (RVAI<sup>17</sup>). The RVAI key can be the key authenticating RIM\_auths to the MTM at the same time. The RVAI is also used to verify other or new verification keys in the named key hierarchy and to authenticate RIM-certificates used to vouch a CounterBootstrap increment request, e.g. for firmware updates. [Mtm, p.21, 46].

---

<sup>14</sup>**flash** or flashing is the process where data or software is transmitted and stored into the permanent memory of a hardware component [Wikc].

<sup>15</sup>**firmware** is a stored software inside the permanent memory of a hardware component [Wikb].

<sup>16</sup>**EK** (Endorsement Key) is a key used by the MTM to sign RIM-certificates 3.2.2 [Mtm, p.8, 29, 82], [Tpm, p.41, 67].

<sup>17</sup>**RVAI** (Root Verification Authority Identifier) is a key used by the MTM to verify new passed verification keys to it.



### **Storage Root Key**

The Storage Root Key (SRK<sup>18</sup>) is used to sign certificates providing new verification keys [Mtm, p.29]. It is sealed inside the MTM and bound to an owner at manufacture time. If the owner changes a new SRK is generated for the new owner [Tpm, p.32].

### **Attestation Identity Keys**

With the Attestation Identity Key (AIK<sup>19</sup>) the MTM signs measured PCR values, vouching the platform's integrity. The signed values are used to verify an intact integrity of the transmitted values and their origin, e.g. to a remote service. A remote service would be able to verify and trust the integrity of the values and treat them as a trusted measurement [Mtm, p.8]. The AIK is an alias for the Endorsement Key [Tpm, p.56].

## **3.3 MRTM - Mobile Remote-owner Trusted Module**

The MRTM is a MTM part configured at manufacturing time to enable remote attestation. Its ownership is set to a remote entity and with it, the urgent secure boot is activated. This configuration enables essential mobile specific commands and optional features. These mobile specific commands include the verification of RIM-certificates and their checks. The installation of RIMs and the ability to load keys and maintaining processes for the MTM. Verifying a RIM-certificate is sufficient to trust into the RIM content because only RIM\_auth parties can create genuine RIM-certificates and will only do that for trusted content, e.g. for a tested program with no malicious behaviour. A verified RIM-certificate extends the corresponding RIM in a specified PCR defined in this RIM-certificate. Loading a new verification key, respectively adding it to the MTM, causes the verification of the given key before storing it in the PCRs. The check can either be performed using a RIM-certificate, or directly using the hash and signature verification. The latter is usually used during the setup of the MTM at manufacturing time, especially when assembling the RVAI. The RVAI can be sealed in a PCR by the MRTM to shield it against manipulations [Mtm, p.13-15, 38, 62].

## **3.4 MLTM - Mobile Local-Owner Trusted Module**

The ownership differentiates the MLTM (local-owner) from the MRTM (remote-owner). A MRTM has to provide the functionalities mentioned above, especially the secure boot. The MLTM can

---

<sup>18</sup>SRK (Storage Root Key) is a key used by the MTM to sign other keys used by MTM.

<sup>19</sup>AIK (Attestation Identity Key) is an alias for the EK [Tpm, p.68].

provide them optionally. The MLTM is considered to provide security for local access, similar to the TPM for PCs. It can be set up from discretionary to mandatory security. The latter would be equivalent to the MRTM setup, except for the owner attestation. For this verification, an endorsement key (EK) must be specified for the MLTM. This key is used to register certificates for verification keys. To authorise the device user to the MLTM a TPM owner authorisation data is used, e.g. a password. After this data is verified, the user can perform commands with the MLTM, e.g. let the MLTM create a RIM-certificate for a program.

The MLTM is allowed to enable, change, or delete the actual set ownership. In Contrary the MRTM does not provide this. Without a defined owner, the MLTM can accept a special RIM-certificate allowing the device into boot to a state where the ownership can be set [Mtm, p.13-15, 38].

## 4 Linux Kernel

This chapter covers the parts of the linux<sup>1</sup> kernel, which are relevant for the proposed security architecture.

The linux kernel is the core part of every linux operating system (OS). It provides services to all other parts of the OS. Usually, the kernel handles the available hardware and their interrupts<sup>2</sup>. It provides a scheduler to assign CPU<sup>3</sup> time and resources to programs, has a memory-management-system to manage the memory assigned to programs and provides many other different system services. The kernel is the base component enabling the execution of code (programs), managing their execution and scheduling<sup>4</sup>. The linux kernel is executed in one address space, the kernel-space. This address space represents a virtual memory<sup>5</sup>, pretending a coherent memory “block” with the address starting at 0. All kernel functions and components share this memory. This virtual memory is protected against access from outside running programs. The entire operating system is running inside the kernel-space and provides a high-level virtual interface of the underlying hardware to the currently running programs. This described characteristic is called monolithic kernel. A secured-memory-management-unit<sup>6</sup> provides the memory management for the kernel’s virtual memory. This secured-memory-management-unit manages the access to the virtual memory [Lov05, p.50]. The provided addresses are only addressable from within the kernel-space. The user-space is protected of accesses of programs executed in non-kernel-space. The user-space virtual memory is managed by a memory-management-unit. Similar to the secured-memory-management-unit it pretends a coherent memory to user-space programs starting at address 0. The addressed “real” memory (RAM<sup>7</sup>) by both memory-management-units can be disordered and fragmented [Lov05, p. 33]. To connect these two address-spaces, the kernel

---

<sup>1</sup>linux is a free open source operating system [Lov05, p.31].

<sup>2</sup>interrupts is a state where the interrupting event perform its task imideatly preempting other actual running tasks [Lov05, p.115].

<sup>3</sup>CPU (Central Processing Unit) is the hardware which processes the instructions of a program [EK11a].

<sup>4</sup>scheduling is a arbitration logic which manages the temporal execution of programs in operating systems [Lov05, p.73].

<sup>5</sup>virtual memory is a possibly phisicly fragmnted memory pretending to processes being coherent and starting with the adress 0.

<sup>6</sup>memory-management-unit arbitrates a virtual memory to the user. It protects the access to each virtual memory [Lov05, p.50].

<sup>7</sup>RAM (Random Access Memory) is a volatile memory which loses its stored information when powered off [EK11b].

provides so called system-calls<sup>8</sup> functions from the kernel-space to the user-space [Lov05, p.102]. Invoking system-calls from the user-space instruct the kernel to perform operations “in the name of” the calling user-space program, e.g. to write to or read from files. The system-call interface prevents the direct interaction and influence of user-space programs to the linux kernel workflow and the hardware.

## 4.1 Device Nodes

The linux kernel abstracts (hardware) devices in terms of ordinary files. All files are accessible through the system-call interface reading information from them or writing information to them if allowed. To interact with the hardware from within the users-space, the kernel arbitrates and uses device nodes. Device nodes are categorised as character or block devices. These categorised devices differs in the accessibility to their content. A character-device can only be accessed sequentially. Reading from and writing to this file is like reading and writing to the PCs serial interface. On the contrary the content of a block device can be accessed arbitrarily, similar to the PCs hard disk. As the device files are “just” abstract representations of device categories, system-calls on them have to be further differentiated. Therefore, the device files are connected with kernel function system-calls used with/on the connected device file. The kernel handles such requests (system-calls) from the user-space with the connected function. According to the kind of device file and the user’s permissions the kernel performs the requested task (or not) and informs the user about the results when finished. The kernel identifies device files by major and minor numbers which are assigned during creation of these special files. The major number identifies the device “family” to identify how to handle a request on a device file matching this major number, e.g. handle a request for a device connected using USB<sup>9</sup>. The minor number identifies the actual device the kernel should perform the request with, e.g. read from the third USB port. From within the user-space, the device files can be accessed by filename. Some (most) device files are located in the `/dev/` directory. Listing this directory with the `ls -l` command reveals some information about the device files. For example:

```
brw-rw-- 1 root disk 8, 0 2011-10-07 10:35 sda
```

and

```
crw-rw-- 1 root dialout 4, 64 2011-10-07 10:35 ttyS0
```

The first letter ‘b’ or ‘c’ identifies the device as a block (b) or a character (c) device file. The letters ‘r’, ‘w’ and the ‘-’ identifies the permissions for groups and users accessing the device

---

<sup>8</sup>system-calls is a set of functions arbitrated by the kernel to the user [Lov05, p.102].

<sup>9</sup>USB (Universal Serial Bus) is a connection standard for various external devices [UIF11].

files. Both files shown are owned by root. The next entry identifies the associated group of the files, i.e. disk for the first and dialout for the latter. The '8, 0' in the first listed device (disk) represents the major, minor number tuple<sup>10</sup>. The '8' represents the device family and '0' represents the actual device attached to the device file /dev/sda. Analogue, '4' is the family of /dev/ttyS0 and the attached device to this file is '64'. The device names (sda and ttyS0) can be chosen freely during creation, as the kernel does not use the filenames but the major and minor tuple for interaction [SBP07, p. 3.1.6]. A device file can be created using the mknod command. The command creating a device file is used as follows:

```
mknod -m permissions name type major minor
```

For sda it would be:

```
mknod -m 60660 sda b 8 0
```

and for ttyS0:

```
mknod -m 20660 ttyS0 c 4 64.
```

Device files can, if permitted, be written to or data can be read from the same way as reading and writing common files in the filesystem. Reading and writing to character or block device files differs in the offset that has to be specified for block devices. The character device can only be read or written at the actual position (pointer) and thus does not need an offset specified. Reading and writing to device files hooks<sup>11</sup> up a system-call performing the requested task by the kernel. This hook results in a kernel-function call attached to the major, minor number tuple (if there is one attached). The `register_chrdev( major, name, fileoperations )` function is used by kernel functions to connect to a device file. As shown above, the major parameter indicates the device family and the name differentiates the device families. The fileoperations parameter is a pointer to a struct (`struct file_operations fileoperations`). This struct contains a function pointer to functions provided by this device family, e.g. write to a device. All operations supported by the device family are listed in this struct. The listed functions represent a subset of or all possible functions supported by this kind of device (block or character device) [SBP07, p. 4.1.1].

## 4.2 Kernel Modules

The linux kernel provides dynamic loading and unloading of kernel modules. Kernel modules are dynamically loadable code providing new functions extending the kernel with additional functionality. With this feature activated, the kernel is able to extend its functionality on

---

<sup>10</sup>tuple is a list of congeneric entires.

<sup>11</sup>hooks describes a point where a specific function is called.

demand without the need of restart. The kernel's ability to load kernel modules is configured and activated at kernel's compile time<sup>12</sup>. Kernel modules must have at least two functions, a start and an end function. The start or init function is called when the module is loaded. It is usually used to initialise the module. To make this entry point be recognisable by the kernel, the `module_init(...)` function call is used with the function-pointer to the init function used as parameter. Analogue, `module_exit(...)` is used for the ending function, called when unloading this module. In addition, these functions are marked with `__init` and `__exit`. Every kernel module has to include the `<linux/module.h>` to be loadable and to arbitrate the named functions. Once loaded, the kernel is able to use all functions provided by this module. Afterwards, the loaded module itself is part of the kernel and shares all functions with and from the kernel [Lov05, p.37, 356], [SBP07, p. 1.1].

### 4.3 Process

A process is a program during its execution. It covers all its claimed resources like e.g. opened files, attached signals<sup>13</sup> and its private address space [Lov05, p.323]. The actual task of a program is performed by threads. Threads are running inside a process and use the resources claimed by the process to perform work. These resources are shared without restrictions between all threads running inside a single process [Lov05, p.66,68]. Each thread has an unique program counter to remember the actual instruction sequence. The program counter enables the thread to continue when the process is scheduled, as described in 4.4.2. Each thread has the pointer to the process stack<sup>14</sup> and the claimed registers<sup>15</sup> of the process. Threads are scheduled by the kernel's scheduler (cf. 4.5) for execution.

#### 4.3.1 Process Creation

The linux kernel creates processes. To create a new process the `fork()` system-call is used, which in turn calls `do_fork()`. Inside `do_fork()` various conditions are tested. If these conditions are passed the calling process is duplicated and the copy is set up with new parameters. The process calling `fork()` is labelled as parent-process while the duplicated process is labelled as child-process. After calling `fork()`, the parent-process proceeds and the child-process executes. The `fork()` system-call returns after successful execution into both processes (parent and child) [Lov05, p.64]. If the child-process is designated to perform a different task than its parent-process,

---

<sup>12</sup>**compile time** is the moment when source code of a program is being compiled into a binary file.

<sup>13</sup>**signals** is a short defined operating system message for processes.

<sup>14</sup>**stack** is a memory where the last set (written) data is the first data received from it when read.

<sup>15</sup>**registers** is a memory storage (hardware) inside the CPU [Wikh].

it has to execute a new (different) program. To avoid unnecessary and expensive copying of data from the parent-process to the child-process for every `fork()` system-call, the kernel uses the copy-on-write<sup>16</sup> mechanism starting to copy the parent-process entries only if the child-process tries to change (write) the initially shared data [Lov05, p.63]. After the copy is done the child-process is a total independent process. Usually, this situation occurs when the child-process does not perform the same task as its parent-process. In this case, the child-process has to be prepared for the new task. The `exec()` system-call then reads the executable binary data of the new program and copies the executable code to the child-process together into a new own address space. Then this executable code (program) is used to start the child-process. Inside the linux kernel, processes are labelled as Tasks.

## 4.4 Task

Tasks are managed by the linux kernel within a doubly linked list<sup>17</sup>. Each element of this list is a process descriptor which contains all information about the associated task, e.g. an unique identifier for each task (cf. Section 4.4.1) or the opened files. This descriptor is called the `struct task_struct` and is described in the `<linux/sched.h>` header file. The actual task is accessible at execution time with the `current` macro, which avoids expensive iterations through the whole list of tasks to reach the task that is currently executed. The `current` task is the actual process currently executing that has been scheduled by the kernel [Lov05, p.56].

### 4.4.1 Process Identification Number (PID)

Tasks are identified by a unique process identification number stored inside their `task_struct`. When the system-call `fork()` is executed, it creates a child-process and assigns a new unique PID to its `task_struct` (`task->pid`) as well as the pointer to the parent's `task_struct` inside its own `task_struct` (`task->parent`). This enables to directly inform the parent process about the child's process state without any overhead imposed by searching for the parent. The pointer to the child's `task_struct` is added to the parents children list (`task->children`). This enables to e.g., terminate all child processes if the parent process has terminated, again, without much overhead. The size of the datatype `pid_t` used for the PIDs thus represents the maximum number of coexistent tasks inside the running system. The PIDs can be reused after a

---

<sup>16</sup>**copy-on-write** This mechanism is used to prevent unnecessary copy operations of one process to the other. Only if the child process tries to write on the address space shared by parent and child a copy operation is performed [Lov05, p.63].

<sup>17</sup>**doubly linked list** Is a chained data structure where every element is connected with its predecessor and its successor.

task has ended its work. Usually the assigned PIDs are simply counted upwards one by one. For this reason, PIDs are only reused after the maximum number, representable by a `pid_t`, for a PID has been assigned during runtime and when another PID is required [Lov05, p.58].

#### 4.4.2 Process State

The actual state of the task is stored inside the state field (`task_struct->state`) of this process [Lov05, p.59]. Each task can be in any of the following five states:

**TASK\_RUNNING** The task is currently running, or waits to be scheduled by the kernel.

**TASK\_INTERRUPTIBLE** The task sleeps and waits for a certain event or signal to wake up. On such an event, the task's state is set to **TASK\_RUNNING** by the kernel.

**TASK\_UNINTERRUPTIBLE** This state is similar to **TASK\_INTERRUPTIBLE** with the difference, that a signal cannot wake up the task.

**TASK\_ZOMBIE** After a task calls the system-call `exit()`, which releases all resources gained by this task, it switches to this state. The `task_struct` has to stay inside the doubly linked list until its parent-process calls the system-call `wait()`. This is necessary, because the parent-process must be informed about the `exit()` "event". After the parent-process's `wait()` call, the child-process `task_struct` is removed from the list.

**TASK\_STOPPED** In this state, the task's execution has ended. The task cannot be executed again. This state is entered after the task receives signals from the debugger or other stop signals.

### 4.5 Thread Scheduling

The kernel's scheduler is a subsystem managing the allocation of resources and CPU time for the tasks. This scheduler allows multitasking pretending concurrent execution of programs to the user on a single CPU and it allows the usage of multiple CPUs to execute threads. It solves the difficulty of the opposed goals of maximum utilisation and fair resource sharing. It tries to solve situations where a task is never executed (starvation<sup>18</sup>). The scheduler employs policies to calculate a fair resource dispersion to a task, containing CPU time and memory. It calculates the priority of each task asking for execution and reduces its priority again after the task has been executed

---

<sup>18</sup>**starvation** is the situation where a task is always preempted from execution by other tasks.



for a period of time. The calculated priority is stored into the `task_struct->prio` and used, besides other values like `task_struct->sleep_avg` indicating the time the `task->state` was `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`, to calculate the at next task to be executed [Lov05, p.73].

## 5 Android Operating System

The security architecture is elaborated to be used with the Android OS. The decision to use Android OS for the development and the evaluation of the security architecture is based on the fact that Android OS runs on a linux kernel. It is an open source operating system and free to obtain and used in the major part of mobile devices on the mobile market.

The Android operating system is developed by Google<sup>1</sup> for mobile devices. It is an operating system tailored to be able to run on low-end resource-limited devices, compared to actual desktop PCs and current Laptops. The initial version of Google's Android OS was developed for systems having a CPU running at less than 500MHz, 64MB RAM at max. and without available swap space<sup>2</sup>. Android is build on top of a linux kernel, version 2.6.35. The kernel is extended with mobile and android specific modules and drivers. Figure 5.1 shows the system architecture of the Android OS. The following sections shortly describe the components of Figure 5.1 in terms of the proposed security architecture. They are described upside down starting with the kernel [Goo11c].

### 5.1 Android OS Kernel

Besides other additions to the kernel, such as mobile specific drivers and a more strict power management module compared to the linux implementation, the Android OS kernel implements a component for inter process communication<sup>3</sup> (IPC) for its own, the Binder. The linux genuine IPC component transmits data objects, containing the whole function range of the requested operation, between the interacting processes. The object and its functions are then executed and used on the requester side. Contrary, the Android OS IPC Binder acts as a proxy<sup>4</sup> by transmitting the requests and afterwards the computed results, between the communicating processes. The computation of the results are performed by the process which was requested for the operation.

---

<sup>1</sup>google is an internet service provider and a software development company [Goo11b].

<sup>2</sup>swap space is a memory available on a non volatile storage. It is used to temporary store data from RAM to gain free memory for other purposes/tasks.

<sup>3</sup>inter process communication is method utilized by processes and threads to communicate with each other. It is used to pass information, requests and results [Wikd].

<sup>4</sup>proxy is a mediator service between different entities..

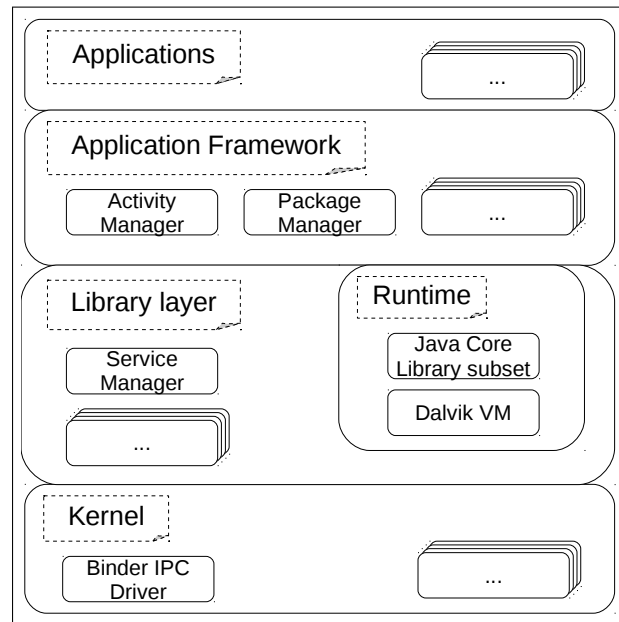


Figure 5.1: Android OS system architecture with its consisting separated layers

The Binder, designed for low-end resource-limited devices, is an essential part of the Android OS. It lowers the memory consumption compared to the linux's own IPC and enables Android OS to share Android application parts and functions between different Android applications. This characteristic also enables to use Android OS specific shared libraries<sup>5</sup> efficiently in Android applications. These libraries are designed to arbitrate an efficient handling of memory usage and computation for essential functions such as displaying a list of elements on the display, e.g. a list containing contact entries [Bra08a], [Bra08b, sld.8].

## 5.2 Android OS Library Layer

On top of the kernel, Android OS provides a layer of libraries written in the C/C++ programming language. These libraries provide interfaces for functions used by the upper Android Runtime layer (cf. Section 5.3). On the one hand this layer or barrier provides a consistent interface to the upper layers (and to the developers) and on the other hand it separates the GPL<sup>6</sup> licensed parts of the kernel from these layers. This enables proprietary closed code to be used and distributed for the Android OS. Beside other libraries, the library layer provides the Service Manager. The

<sup>5</sup>libraries is a loadable commulation of additional functions loaded at runtime by a program.

<sup>6</sup>GPL (GNU General Public License) forces programs/libraries using code written under GPL also to be released under this license [FSF11].

Service Manager allows to share services between different applications. It closely interacts with the Binder module [Bra08a], [Bra08b, sld.32-53].

### 5.3 Android OS Runtime

The Android OS runtime is located inside the library layer. It implements the most features of the JAVA programming language (core libraries) and uses the above described Android OS libraries to provide and arbitrate Android OS features to the upper layers. Another important part of the Runtime layer is the Dalvik Virtual Machine<sup>7</sup>. Every Android application runs in its own process running its own instance of the Dalvik Virtual Machine [Bra08a], [Bra08b, sld.55-58].

#### 5.3.1 Dalvik VM

The Dalvik Virtual Machine (DVM) was developed for the Android OS to be efficient on low-end mobile devices, and to be able to run multiple instances of it on low-end mobile devices as outlined above. It was designed to run on the RISC<sup>8</sup> architecture platform like ARM<sup>9</sup> based platforms. It encapsulates each process within an individual Dalvik Virtual Machine. This encapsulation ensures that each process, inside its virtual environment, is protected from other processes within their virtual environments. It is also intended to protect the Android OS against misbehaving processes and to guarantee stability despite the existence of misbehaving processes. The DVM executes .dex (Dalvik executable) files. These files contain compiled and optimised JAVA bytecode<sup>10</sup> for Android OS [Bor08].

### 5.4 Android Application Framework

The Application Framework is located on-top of the library layer. It exposes the library layer's functions to the JAVA programming language. This layer represents the development API. It provides services to applications and manages their life-cycles. Finally, it provides access to the lower level (library layer 5.2) hardware APIs [Goo11a] [Bra08b, sld.61-75].

---

<sup>7</sup>**Virtual Machine** is a virtual computer with software representations of hardware components [Wikk].

<sup>8</sup>**RISC** (Reduced Instruction Set Computer) describes a CPUs design using a set of simple instructions [Wiki].

<sup>9</sup>**ARM** is an abbreviation for a Arcon RISC Machine CPU.

<sup>10</sup>**bytecode** is a collection of instructions for a Virtual Machine [Wikk].

## 5.5 Android Applications

Android applications are written in the JAVA programming language and compiled into the dex format executable by the DVM (cf. Section 5.3.1).

## 5.6 Android Boot Sequence

This Section describes the boot sequence of the Android OS. It is depicted in Figure 5.2 and illustrates the boot sequence until the first application can be started inside the Android OS. The boot sequence of the linux kernel is shown at step ①. It is assumed that all pre-procedures have already been performed to start up the kernel, such as a battery or other checks. The following bullet-points ① to ⑬ describe the performed step(s) during the boot sequence of the Android OS. They are associated with the corresponding points depicted in the Figure [Bra08b, sld.80-94].

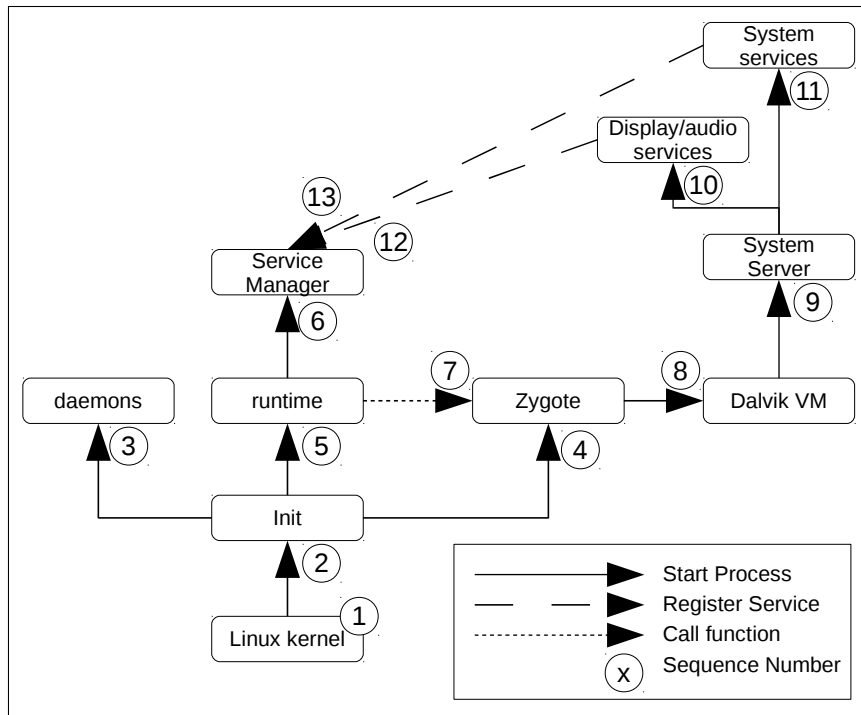


Figure 5.2: Android OS boot sequence (cf. [Bra08b, sl.87]).

- ① Starting a device having the Android OS installed executes the bootloader at first. The bootloader starts the kernel
- ② The kernel starts the first linux process INIT.

- ③ INIT then starts common low-level daemons<sup>11</sup>, typically used to provide low-level hardware services such as the USB device daemon (usbd). It also starts some Android specific daemons such as the android debugger daemon (adb)
- ④ Subsequently, INIT starts the Zygote<sup>12</sup> process. This process initialises the DVM and listens on sockets<sup>13</sup> for requests to spawn further DVMs. If such requests arrive, the Zygote process `fork()`s itself (cf. Section 4.3.1) and provides the `fork()`ed DVM to the requester. This prevents recreation of a new DVM from the ground up every time when an Android process requests to be executed.
- ⑤ After Zygote starts, the INIT process starts a runtime process which is used to establish the IPC service, i.e. the Binder
- ⑥ The runtime process starts the Service Manager, which manages all services accessed and used through the Binder.
- ⑦ Afterwards, the runtime calls Zygote to start the system server process.
- ⑧ Zygote `fork()`s a new DVM
- ⑨ and starts the System Server process inside this DVM.
- ⑩ The System Server starts the display and audio services for the device
- ⑪ and it starts all system services from the application framework (cf. Section 5.4).
- ⑫ The display and audio services are registered into the Service Manager to be accessible by any other process through the Binder.
- ⑬ Equally the System services are registered into the Service Manager.

Finally the Android OS is ready to start a new activity (Android application). The System Server shares the loaded services with all other upcoming Android applications.

---

<sup>11</sup> **daemons** is a service running in the background of the operating system.

<sup>12</sup> **zygote** is a core process in the Android OS which forks new Dalvik Virtual Machines to start new Android processes (Activities).

<sup>13</sup> **sockets** is a data communication endpoint for exchanging data between processes [Wike].

### 5.6.1 Android Application Start

The application start is illustrated in Figure 5.3. It shows the sequence how the Android OS starts a new Android Application. It is assumed for this figure that the Android OS has been booted as described above (cf. Section 5.6). Moreover, Figure 5.3 shows the Activity Manager within step ①. The Activity Manager is a module which initiates new Activities (starting new applications). The Package Manager manages all installed and available Android applications and handles all available data about them. The following bullet-points ① to ⑥ describe the performed step(s) for an Android application start in the associated point depicted in the Figure 5.3.

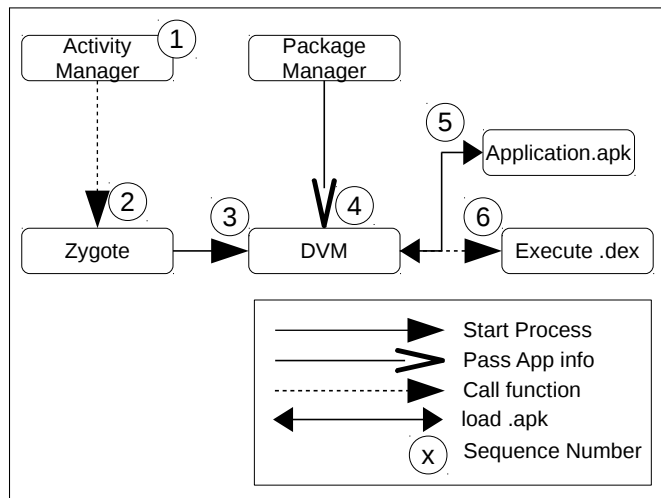


Figure 5.3: Android OS application start

- ① A request to start a new Android Application is send to the Activity Manager,
- ② which in turn sends a request to Zygote to `fork()` a new DVM
- ③ The new DVM is then `fork()`ed by the Zygote
- ④ The information about the `.apk` file corresponding to the requested Android application start is provided by the Package Manager
- ⑤ The Android application file is loaded from the file-system and extracted
- ⑥ The containing `.dex` (Dalvik executable) is started inside the new DVM

## 6 Security Architecture

This chapter discusses the proposed security architecture in detail. It illustrates the functionality and work-flow of the components of the security architecture. The communication and data flow between the components are described as well. The security architecture is divided into small application modules. The implementation of those modules are described in Section 7.

This security architecture closely follows the assumptions and propositions made in Section 2. The hardware requirements listed in Section 2 are covered by the MTM (cf. Section 3). Although no MTM has been manufactured until now it is expected that MTM hardware implementations will possibly emerge in the near future. The MTM is set up at manufacture time to MRTM mode (cf. Section 3.3). This set-up provides remote ownership which cannot be changed afterwards. The MRTM mode allows to the MTM to sign RIM-certificates. Administrators are vouched with a secret shared with the MTM, that ensure them to exclusively be able to request operations from the MTM. After this set-up at manufacture-time, the MRTM performs a secure boot (cf. Section 3.1) every time whenever the device is activated. The secure boot meets the requirements of Section 2.2 and ensures the device's integrity. As the secure boot checks the device's integrity at the boot time, malicious activities and other threats occurred during runtime cannot be detected by the MTM until the next secure boot. Thus, the mobile device's security has to be maintained also during the runtime. A continuous scan and remote-integrity verification using an Integrity Measurement Architecture (IMA) [Sai+04] is not a satisfying option for mobile devices, due to the high computation and energy demands. The same applies for anti-malware and cognate programs. Instead of scanning for malicious programs and activities, which could be considered as the scan for black-list<sup>1</sup> entries, a probably less expensive approach is to allow the execution of only trusted programs, which could be considered as a white-list<sup>2</sup> verification. Programs not listed in the white-list are prevented from execution. An entry within the white-list consists of a RIM-certificate (cf. Section 3.2) of the accepted program signed by the MRTM. This white-list respectively trusted application list (TAPL<sup>3</sup>) is maintained by the administrator of the mobile

---

<sup>1</sup>**black-list** is a list of entries indicating not accepted entries.

<sup>2</sup>**white-list** is a list of entries indicating accepted entries.

<sup>3</sup>**TAPL** (Trusted Application List) contains all calculated and signed hash values of trusted applications.



device. Corresponding to Section 2.3, the administrator can request the MRTM to create a RIM-certificate for the program with a previously calculated hash (i.e. a reference hash value) of this program. Before a program is executed, its hash value, computed at loadtime, is verified against the RIM-certificates of the TAPL and only executed if a corresponding RIM-certificate exists in it and the hash matches. According to Section 2.5, the MRTM's functionality needs to be extended to perform such runtime verifications. Additionally, the operating system, in this case the Android OS, has to be enhanced to use the MRTM's attestation abilities. Every request for a program execution hooks up an attestation. Execution of a program leads to creation an execution of a new process (cf. Section 6.2.1). The new process is created as a child of the calling process. Another function is hooked to verify the calling process. This helps to avoid programs slipped through the first attestation to create child processes. To prevent the execution or even kill<sup>4</sup>, a slipped-through program, the resource scheduler (cf. Section 4.5) hooks the same function to verify a process before arbitrating resources to it. As stated in Section 5.1, the Android OS is based on the linux kernel, but with additions and changes and Android OS specific libraries performing low-level functions such as the Service Manager and the Binder module. All processes are created by the kernel. Android applications are executed inside running DVMs. To secure the process creation, the attestation functions are located inside the kernel and partly inside the Android OS Runtime layers and the Dalvik Virtual Machine code. Both attestation points are verified and protected by the secure boot (cf. Section 3.1). The secure boot prevents an undetected manipulation of the systemfiles, to bypass the verifications. Android applications started in DVMs which are not handled by the kernel. They are handled by the Android OS which provides functions to load and extract the corresponding Android Application file and start it inside the DVM (cf. Section 5.6.1). The attestation, verification, management as well as interfaces for the security architecture are located and implemented mainly as kernel modules (cf. Section 4.2) and partly outside the kernel. The Android OS DVM code is extended with functions required to interact with these kernel modules when attempted to start an Android application. The DVM always obeys the decision of the security architecture kernel modules. Native process creation is fully handled by the kernel and verified directly by the security architecture modules. Hence, the security architecture protects all possible process creation mechanisms on the Android OS.

---

<sup>4</sup>kill is a signal sent to a process causing the process receiving it to stop its task imideatly and to release its gain ressources.

## 6.1 Architecture Modules

The security architecture is based on Google's Android 2.3.3 operating system running on an Android Linux kernel 2.6.35. Figure 6.1 illustrates the modules of the security architecture and their interaction inside of the Android OS. The following part of this section describes the function of the modules. The Section 6.2 describes their interaction.

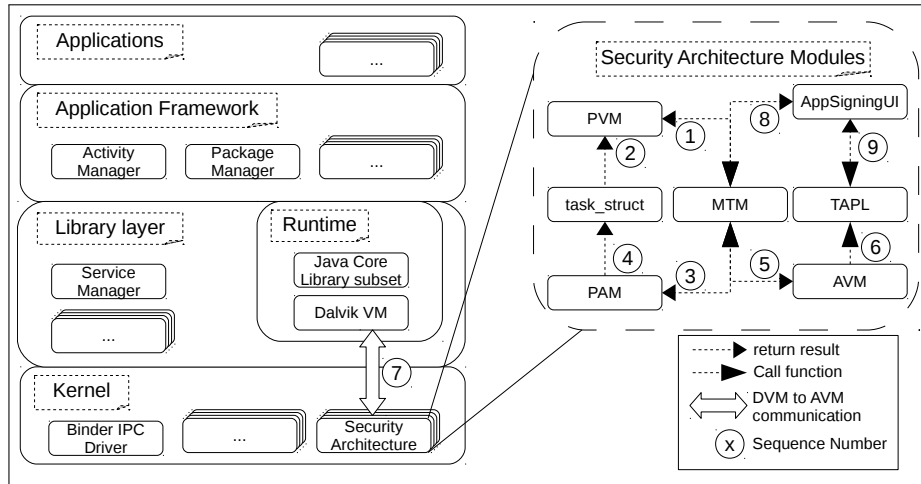


Figure 6.1: Security architecture modules location and interaction

### 6.1.1 Security Architecture Keys

- $Key_{MTMAuth}$  is used in the proposed security architecture to authenticate requests from an administrator to the MTM.
- $Key_{HMAC}$  is used for authentication and verification by PAM and PVM.
- $Key_{PubSig}$  is used by AVM to request the MTM to verify a hash against the signed TAPL entries.
- $Key_{PrivSig}$  is used to sign a given hash value. This key is private and exclusively usable and accessible by the MTM. It is sealed inside the MTM and never released.

### 6.1.2 MRTM Wrapper (MTM)

As there is currently no MTM hardware module available, this wrapper provides interfaces used by the security architecture to decouple it from a real hardware MTM. This kernel module is

loaded at power-on time together with the operating system. Once loaded, it acts as a substitute for a real MRTM. After the secure boot (cf. Section 3.1), its task is to make keys from the key hierarchy (cf. Section 3.2.2) accessible from within the kernel. It also provides cryptographic functions for the signing and verification mechanisms.

### 6.1.3 Trusted Application List (TAPL)

RIM-certificates are managed by this module. It provides functions to add a RIM-certificate to the TAPL, to remove one from it and to handout the whole TAPL (all RIM-certificates) to a requesting entity, e.g. to a kernel module.

### 6.1.4 Application Verification Module (AVM)

This kernel module verifies the computed hash value of a program against the RIM-certificates listed in the TAPL. It requests the MTM to verify the calculated hash value against the RIM-certificates and returns an error code if the verification fails or a success code otherwise.

### 6.1.5 Application Signing User Interface (AppSigningUI)

This is the interface between the kernel and the user-space for the administrator. It is connected to a character device (cf. Section 4.1) and passes signing requests to the MTM which contains the hash of the program and the administrator's shared secret<sup>5</sup> ( $\text{Key}_{\text{MTMAuth}}$ <sup>6</sup>). The MTM returns the RIM-certificate if the authentication with the  $\text{Key}_{\text{MTMAuth}}$  succeeds, which is then added to the TAPL. It also manages requests for getting the content of the TAPL.

### 6.1.6 Process Authentication Module (PAM)

The PAM computes the HMAC using the PID stored in the process' `task_struct`. The `task_struct` (cf. Section 4.4), i.e. the description of every process in linux, is endorsed with the computed HMAC. PAM computes the HMAC using a secret key ( $\text{Key}_{\text{HMAC}}$ ) stored in the MTM.

### 6.1.7 Process Verification Module (PVM)

This kernel module is used to verify the HMAC of a process. Like the PAM, it computes the HMAC using the PID of the process stored in the `task_struct`. This HMAC is compared with

---

<sup>5</sup> **shared secret** is a secret known to all participating communication endpoints, such as a password or a secret key.

<sup>6</sup> A better option is sending a request authenticated with the shared secret instead of sending the Key itself

the HMAC endorsed to this process by the PAM. If the HMACs match, a success code returns, otherwise it returns an error code. A successful verification of the endorsed HMAC indicates, that the process is allowed to be scheduled as well as to create child processes.

## 6.2 System Integration

### 6.2.1 Process Creation

The `do_fork()` function is called (cf. Section 6.2.1) and just before the current process is cloned the PVM is hooked. It requests the `KeyHMAC` from the MTM (cf. Figure 6.1 ①) and computes an actual HMAC from the PID of the process `task_struct` using the `KeyHMAC` (cf. Figure 6.1 ②).

**Success** The PVM compares the computed HMAC with the endorsed HMAC and returns a success code if they match. After this a new process is cloned, the child. The PAM is hooked then and requests the `KeyHMAC` (cf. Figure 6.1 ③) from the MTM to compute the HMAC for the new process and endorses the child's `task_struct` with the computed HMAC (cf. Figure 6.1 ④).

**Failure** The PVM compares the HMAC with the endorsed HMAC and returns an error code. The `fork()` call returns an error code and terminates.

### 6.2.2 Boot Sequence

It is assumed that the MTM has successfully completed its secure boot (cf. Section 3.1) up to the state ⑧ in Figure 3.1. The modules of the security architecture are loaded into the kernel using the `insmod` command. When the kernel has finished loading, the first `do_fork()` is called for the `INIT`<sup>7</sup> process, which in turn calls `do_fork()`. The PVM is hooked here (cf. Section 6.1.7). PVM identifies the `INIT` process, as it is the first process starting after the boot and its PID is 1 and skips its verification. PAM computes and endorses `INIT` with a HMAC (cf. Figure 6.1 ③ and ④) and returns a success code. No further verification is skipped. It continues with the process creation (cf. Section 6.2.1). This procedure repeats until the operating system is fully loaded.

### 6.2.3 Starting a new Program (Linux)

The operating system wants to start a new program. A new process is created as described above in Section 6.2.1. Then the `exec()` system-call is called assuming that the new process performs a different task than its parent. The hash value of the program binary data is computed and the

---

<sup>7</sup> `INIT` is a process directly “constructed” operating system. In this described case the OS is verified and trusted.

AVM is hooked. It requests the  $\text{Key}_{\text{PubSig}}$  from the MTM (cf. Figure 6.1 (5)). It requests the TAPL from the TAPL module (cf. Figure 6.1 (6)) and calls the MTM to verify each TAPL entry using the  $\text{Key}_{\text{PubSig}}$ .

**Success** The verification succeeds and returns a success code. This continues the `exec()` call resulting in starting the new program.

**Failure** The verification fails and returns an error code to the OS, resulting in the termination of the program execution.

### 6.2.4 Starting a new Application (Android OS)

An Android Application is intended to be executed. This intent is passed to the Android OS which initiates the start of a new application. A new process (DVM) is created as described in Section 6.2.1. The DVM receives the information about the Android Application that is intended to execute. The DVM computes the hash value of the Android Application binary and passes the value to the AVM (cf. Figure 6.1 (7)). The AVM requests the  $\text{Key}_{\text{PubSig}}$  from the MTM (cf. Figure 6.1 (5)) and requests the TAPL from the TAPL module (cf. Figure 6.1 (6)). Then it calls the MTM to verify each TAPL entry using the  $\text{Key}_{\text{PubSig}}$ .

**Success** The verification succeeds and returns a success code, such that the DVM is allowed to start the application.

**Failure** The verification fails and returns an error code to the Android OS, such that the Android application start terminates.

### 6.2.5 Administration

The administrator communicates with the security architecture using the character device described in Section 6.1.5. The administrator sends the requested operation together with the  $\text{Key}_{\text{MTMAuth}}$  to this device, which passes it to the AppSigningUI module. Then the MTM is called by the AppSigningUI to verify the administrator using the passed  $\text{Key}_{\text{MTMAuth}}$  (cf. Figure 6.1 (8)).

**Failure** The verification fails and returns an error code. The AppSigningUI module passes the error code to the character device. The administrator receives the error code from it.

**Success** The verification succeeds and returns a successcode. The AppSigningUI continues processing the request (cf. Figure 6.1 ⑨).

### **Receiving the TAPL**

The AppSigningUI requests the TAPL from the TAPL module. Then passes it to the character device. The administrator reads the TAPL from it.

### **Signing and Adding the signed Value to the TAPL**

The AppSigningUI module reads the hash value from the character device, then requests the MTM to create a RIM-certificate. The MTM passes it to the AppSigningUI module. Then the RIM-certificate is passed to the TAPL module, which adds it in its list.

### **Removing a Signed Value from the TAPL**

The AppSigningUI module reads the hash value from the character device and requests the TAPL module to remove a matching RIM-certificate from it.

## **6.3 Security Analysis and Discussion**

- The MTM is the trust anchor of this security architecture. It provides a trusted state of the system, which cannot be compromised in an undetected manner. All other components of this architecture rely on this trust anchor. According to the MTM specifications and partly because it is a hardware based component, it is not feasible to manipulate it successfully. The provided security by the MTM ends up in the secure boot and the ensured trusted state.
- The PAM and PVM covers the process creation and the scheduling. They allow only trusted processes to be executed or cloned/copied. They cannot access the binary files of the application, from which a process was created. The trust and its verification is derived from the fact, that all processes are descendants of the INIT process that is directly trusted. As only verified processes are allowed to create child processes and to be scheduled, no untrusted or manipulated processes should be runnable on the operating system. Processes are inspected by the PVM of their trust and its children then authenticated by the PAM for this reason.

- The AVM is to verify the integrity of programs' binary files at runtime and is consulted whenever a new program or process tries to execute.
- The AppSignigUI is used to perform administrative tasks on the device. Its security depends on the  $\text{Key}_{\text{MTMAuth}}$  (shared secret). A lost or stolen  $\text{Key}_{\text{MTMAuth}}$  compromises the security of the whole architecture, as an attacker would get administrator privileges with this key.

The proposed security architecture secures a mobile device at boot, during runtime until the device is shut down. Its security depends on the discipline of the administrator(s). This security architecture assumes that a program which is installed at the first time is not infected with a malicious code. Once it is installed, its subsequent infections with a malware is detected by the proposed architecture and the infected programs are prevented from execution. This architecture cannot tell apart if a program or process has malicious behaviour or if it is exploitable. Hence, careless signing of untested and untrusted programs or Android applications can compromise the security architecture.

## 7 Implementation

This section presents the implementation of the described security architecture. It closely follows the module descriptions from Section 6 and partly ties up with [UW11] and [UWL11]. The implementation of the security architecture has been done on an Android OS, version 2.3.3 (Gingerbread) running on an Android linux kernel, version 2.6.35. As denoted in Section 6, the Dalvik Virtual Machine located in the Android OS Runtime layer has been modified and the kernel has been extended with the security architecture modules/functions. Besides the character device file for the AppSigningUI module, an extra character device file has been created for the communication between the kernel and the Android OS DVM. The execution of an Android application in a `fork()`ed DVM (cf. Section 5.6.1 Figure 5.3 ③) is performed inside the Android OS. The Android OS DVM is extended with a security hook to a function that computes a SHA256<sup>1</sup> value from the .apk (Android Application binary) file which is about to be executed and communicates with the kernel to check whether this application is trusted or not (cf. Section 6.2.4).

### 7.1 Kernel Modifications

The modifications are corresponding to the workflow described in Section 6.2. The kernel modifications related to the program authentication and verification are:

- `kernelRoot/kernel/fork.c`  
The `do_fork()` method of this file is modified. Applications are verified and authenticated using `KeyHMAC` using PAM and PVM.
- `kernelRoot/include/linux/sched.h`  
The HMAC authenticating a process is stored in the `struct task_struct->ucaHMAC`. This extension self manages the HMACs of each task as no list have to be maintained and managed to store and verify the HMACs.

---

<sup>1</sup>SHA256 (Secure Hash Value) is a cryptographic hash representing a unique checksum of 256 bit length [MOV01, ch.9.52].



- `kernelRoot/arch/arm/kernel/sys_arm.c`  
Native applications are verified within `fork()` and `exec()` system-calls via PVM, PAM modules during `fork()` and via AVM during `exec()`.

Additionally, kernel modules are used to perform the following tasks for this security architecture:

- `sec_autoload.ko`  
loads and parses the `sec_TAPL.txt` from the filesystem which contains the signed SHA256 values of the trusted applications and adds them to the `sec_tapl.ko` module.
- `sec_cryptoavm.ko`  
used by the `do_fork()` call and by the `sec_verify_bridge.ko` to verify signed SHA256s.
- `sec_key_storage.ko`  
receives the keys from the `sec_rsacryptomtm.ko` after the secure boot (cf. Section 3.1) and arbitrate them inside the kernel.
- `sec_rsacryptomtm.ko`  
this is an MTM wrapper interfacing the security architecture with a real MTM hardware. Right now it is implemented as a stub<sup>2</sup> which performs the requests from the security architecture with software elements.
- `sec_tapl.ko`  
this module manages the TAPL entries.
- `sec_ui.ko`  
used to process administrative tasks sent to the security architecture from the user-space. This module is connected to the character device `/dev/sec_device` for this reason.
- `sec_verify_bridge.ko`  
this module is necessary to verify Android Applications. Similar to the `sec_ui.ko`, this module is connected with the character device `/dev/sec_device_vb` to establish a communication between the modified Android OS DVM and the security modules implemented in the kernel.

### 7.1.1 `fork.c` Modifications

The following part describes modifications made on the `fork.c` file located in the `kernelRoot/kernel/` directory. It contains beside other functions the `do_fork()` function which is called from the

---

<sup>2</sup>**stub** is a piece of code acting as a substitute for full implementation of a function or a program.

system-call `fork()` and has been modified to perform the PAM and PVM authentication and verification operations. The modified behaviour of the `do_fork()` call is illustrated in Figure 7.1. The bulletpoints ① to ⑦ describes the steps performed by PAM and PVM during process creation. The Bulletpoint ⑤ represents the authentication failure state resulting in the return of an error code.

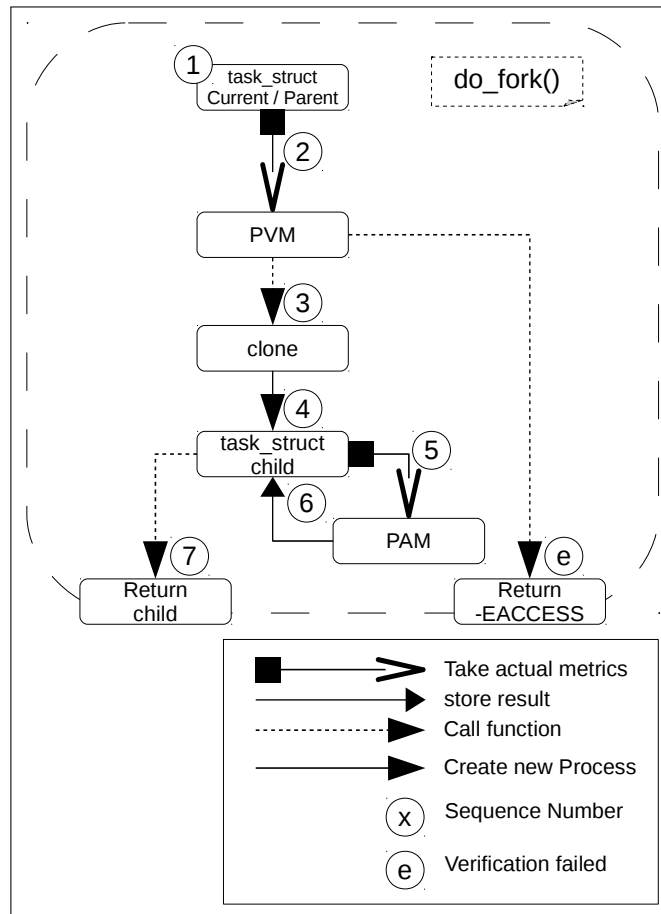


Figure 7.1: Execution of the `do_fork()` method hooked with PAM and PVM checks

- ① A process calls the `fork()` system-call to create a child process. `fork()` in turn, calls the `do_fork()` function of `fork.c`.
- ② Just before the current (caller) process is copied/cloned, its HMAC field `current->ucaHMAC` is verified by the PVM module using the `KeyHMAC` key.

- ③ If the verification succeeds, the `current` process continues with the copying/cloning the `current` process.
- ④ This results in a copied but not running process namely the child process.
- ⑤ The HMAC for the child process is then computed by the PAM module using the `KeyHMAC` key
- ⑥ and set into the child process' `task_struct->ucaHMAC` field.
- ⑦ The child process is created and returned from the `fork()` function to be further used.
- ⑧ At a verification failure of the parent's HMAC (`current->ucaHMAC`), the function returns with `-EACCESS` error code, which terminates the creation process of the child.

The described procedure is identical for all processes except the `INIT` process. `INIT` is verified by the secure boot and assumed as trusted automatically. Its HMAC is calculated and the `current->ucaHMAC` field of `INIT` is directly set by the PAM module. This bypasses step ② and performs steps ⑤ and ⑥ for `INIT` before it continues with ③. The security modules PAM and PVM are directly implemented within the kernel.

### 7.1.2 `sys_arm.c` Modifications

The functions implemented in `sys_arm.c` are the points where all linux process creations start. This file includes, beside of other functions, the `fork()` and `exec()` system-calls. The `sys_arm.c` is extended with the ability to load binary files from the filesystem and to compute SHA256 values over them. This function is used for the binary files which are executed with the `exec()` system-call. The computed SHA256 value is passed to the loaded security module `sec_cryptoavm.ko` (`sec_AVM`) for verification (cf. Figure 7.4 ①). If verified, the process execution is continued otherwise `-EACCESS` (error code) is returned which terminated the execution request.

## 7.2 Security Architecture Modules

### 7.2.1 `sec_tapl.ko` Module (`sec_TAPL`)

This module manages all TAPL entries. As no MTM hardware is available to create genuine RIM-certificates, each TAPL entry consists of the signed SHA256 value and the filename of the corresponding binary file.

### 7.2.2 `sec_autoload.ko` Module

This module reads the `sec_TAPL.txt` from the filesystem and parses the entries to add them to the `sec_tapl.ko`. It uses the interface provided by the `sec_TAPL` module to add these entries to the TAPL. This module depends on `sec_tapl.ko` to be loaded before.

### 7.2.3 `sec_cryptoavm.ko` Module (`sec_AVM`)

This module provides the verification of a given SHA256 value. When the module is loaded into the kernel, it requests the `sec_key_storage.ko` module to hand out the `KeyPubSig` used for the verification. Verifying a given SHA256 value requests the `sec_tapl.ko` (TAPL module) to handout the TAPL content containing the signed SHA256 value of the program (cf. Figure 7.4 (2)). Each signed SHA256 value is then verified against the given SHA256 using the `sec_rsacryptomtm.ko` module (`sec_MTM`) until the given SHA256 value matches one TAPL entry. The result (1 match, 0 no match) of the verification request is returned to the caller. This module depends on `sec_key_storage.ko` and `sec_tapl.ko` modules.

### 7.2.4 `sec_key_storage.ko` Module (`sec_Key_Storage`)

This module provides the keys, `KeyPubSig` and `KeyHMAC`, to the security architecture. Loading this module into the kernel prompts a request to the `sec_rsacryptomtm.ko` (MTM) to reveal the keys. Once stored, the module arbitrate these keys to the other modules. This module depends on `sec_rsacryptomtm.ko` module.

### 7.2.5 `sec_rsacryptomtm.ko` Module (`sec_MTM`)

This is the interface module between the security architecture and a real hardware MTM (cf. Figure 7.4 (4)). It provides a function to let the MTM sign and verify a SHA256 and to reveal the sealed keys from the MTM. When loaded, it first performs an integrity check of the operating system and compares it with the previously MTM computed values by asking the MTM to handout the corresponding RIM. If the checks pass, the module requests the keys from the MTM. Otherwise the module unloads itself. As no MTM is mounted on the device and the implementation was made with no MTM available, the key request and the integrity check are skipped and asumed as performed and successfull. The keys are hardcoded in the `sec_rsacryptomtm.ko` for this reason.

Signing operations are protected using the `KeyMTMAuth` (shared secret) known only by the MTM and the administrator to authenticate the administrator to the MTM. If the administrator is

successfully authenticated to the MTM, this module returns the signed value back to the caller otherwise an error code is returned indicating that no signing operation has been done.

### 7.2.6 `sec_ui.ko` Module (`sec_UI`)

This module is the interface between the administrator and the security architecture. It uses the character device file `/dev/sec_device` to connect with the user-space and provides operations to the administrator to sign a given SHA256 value, to remove entries from the TAPL or to receive the content of the whole TAPL. Communication is done using the IO control system-call `ioctl(...)` on the `/dev/sec_device` file.

An administrative request consists of:

- the secret authentication key for the administrator `KeyMTMAuth`
- a command number indicating the type of request it is:
  - 1 to sign a SHA256 value and include it to the TAPL
  - 2 to remove one entry from the TAPL
  - 3 to receive the content of the whole TAPL
- and finally, data necessary for the request (e.g., the SHA256 value that should be signed and added to the TAPL)

Administrative requests, passed to this module, are `marshall`<sup>3</sup>ed into a special format. This format uses `structs` as container and `enums` for the command numbers, defined in `sec_ui_commands.h` file.

The IO control system-call expects three passed parameters from the user-space:

1. the filepointer to the device file, e.g. `/dev/sec_device`
2. a command parameter of type `unsigned int`
3. and a value parameter of the type `unsigned long`

The pointer to the marshalled request is stored into an `unsigned long` variable and used as the value (third parameter). The `enum` indicating the request type is used as the command (second parameter) and finally the pointer to the `/dev/sec_device` device file is used as the first parameter. Once the request is passed to the module, it unmarshalles the data according to which command number (`enum`) has been passed.

---

<sup>3</sup>`marshall` is the conversion of data structures into a format used to transfer these structures between communicating processes.

### Marshall Format UI

All data sent to or received from the character device (`/dev/sec_device`) is represented by an unsigned `char` array. Together with the size of this array (number of used bytes) it is saved in a `struct stData`. All `stData` structs used for one request, are stored in a `struct stParameterContainer` together with their quantity. The type of the data is only identified by the passed command (enum), indicating the type of the request.

#### 7.2.7 `sec_verify_bridge.ko` Module (`sec_Verify_Bridge`)

This module is used to establish the Android Application verification from inside of a DVM. Similar to `sec_ui.ko`, this module is connected to a character device file (`/dev/sec_device_vb`). It is used to verify an Android application just before it is loaded into the DVM. The SHA256 value of this Android Application file (.apk) is marshalled together with a pointer to the variable `cCheckPassed` which is used for storing the verification result. The marshalled data is then passed to `/dev/sec_device_vb` using the system-call `write(...)`. If something is written to this module, it locks<sup>4</sup> the kernel function and disables all interrupt requests (IRQ<sup>5</sup>s). This avoids a race condition between the Android process start in the DVM and the verification in the kernel. Then it unmarshalles the information and asks the `sec_cryptoavm.ko` (`sec_AVM`) to verify the given SHA256 value. The result is stored inside the `cCheckPassed` variable. Subsequently the module releases the lock and restores (re-enables) all IRQs [Lov05, p.173, 192].

### Marshall Format VB

This format is a `struct stVerificationSHA256` containing an unsigned `char` array with the size of 32 bytes for a SHA256 value and a unsigned `char*` pointer pointing to the result variable `cCheckPassed`.

## 7.3 Android DVM Core Modifications

The Android OS Dalvik Virtual Machine is extended with a verification function. This function is hooked into the `Dalvik_dalvik_system_DexFile_openDexFile(...)` function, located in the file `androidOS/dalvik/vm/native/dalvik_system_DexFile.c`, just before an Android

---

<sup>4</sup>**locks** is used by the kernel to secure critical sections. Only one process can have the lock and enter the critical section. Other processes are forced to wait to enter the critical section until the process possessing the lock leaves this section and releases the lock.

<sup>5</sup>**IRQ** an interrupt request initiates an interrupt which cause the actual running task in the CPU to suspend its work and to let the interrupting task perform its work.

Application is loaded from filesystem (cf. Figure 7.4 ⑤). The verification function loads the binary file, calculates its SHA256 value and put it together with the pointer to the variable `cCheckPassed` into a struct `stVerificationSHA256`. The value of `cCheckPassed` is set to 0 by default which indicates a failed verification. The passed pointer to this variable is used to copy the verification result from within the kernel to the `cCheckPassed` variable. The marshalled data (`stVerificationSHA256`) is then written to the character device file `/dev/sec_device_vb` connected with the `sec_verify_bridge.ko` module (cf. Figure 7.4 ⑥). The value of `cCheckPassed` is then returned from the verification function performed by the `sec_AVM`. If the verification succeeds (returned 1), the Android Application is loaded and executed inside the DVM, otherwise an exception<sup>6</sup> (`IOException`) is thrown to the Android OS from within the DVM, which terminates the loading of this application.

### 7.4 Implemetation Workflow

In this section the workflow of the security architecture is described in detail. First, a description of the security operations performed in the linux kernel is made. Subsequently, a description of the security architecture from the Android OS point of view is discussed.

#### 7.4.1 Workflow for the Linux Kernel

Figure 7.2 illustrates the workflow in the linux kernel. The PAM is labled as `sec_PAM` and the PVM as `sec_PVM` in this figure. Steps ① to ④ are performed during the boot process just before the INIT process is started. The `sec_autoLoad.ko` module is not shown in the figure. It is asumed that this module is loaded just after the `sec_TAPL` and fills the TAPL as specified in Section 7.2.2.

- ① The MTM performs a secure boot until the kernel is executed by the bootloader. The corresponding entity `(MTM)` is faded out, as no MTM hardware is available and the implementation have been done without such hardware. However, it is assumed that such hardware will emerge soon in the mobile device industry.
- ② Firstly, the security module `sec_MTM` is loaded,
- ③ Secondly, the `sec_TAPL` module and just after it the `sec_autoLoad.ko` module, which reads the `sec_TAPL.txt` file from the filesystem. `sec_TAPL.txt` file contains the signed

---

<sup>6</sup>**exception** is an information about a process state. An exception is “thrown” to other application layers [Lov05, p.116].

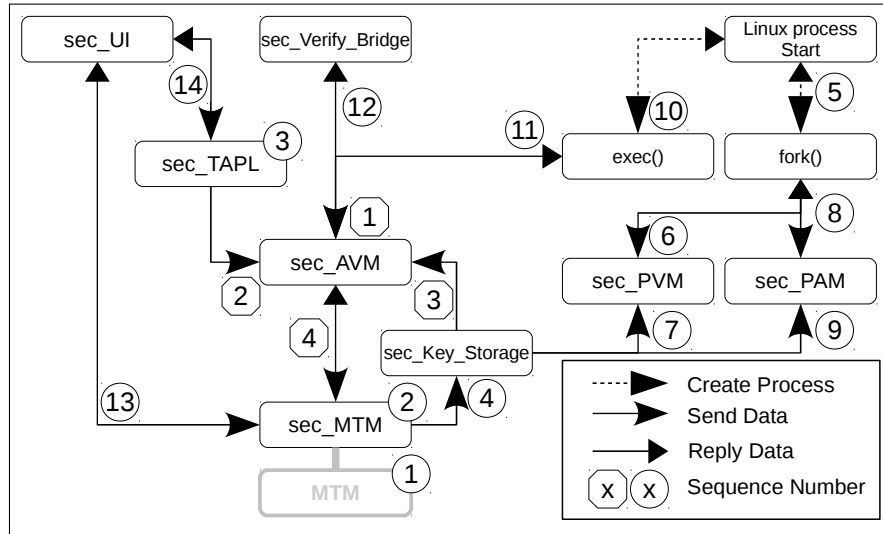


Figure 7.2: Overall Security Architecture Workflow

SHA256 entries and the filenames that belongs to them. `sec_autoload.ko` module parses this file and adds the parsed entries to `sec_TAPL` using the interfaces provided by `sec_TAPL.ko`.

- ④ Thirdly, the `sec_Key_Storage` module is loaded. It requests the `sec_MTM` to handout the keys, `KeyHMAC` and `KeyPubSig`, to provide them to the other modules. After this step, all remaining modules of the security architecture are loaded into the kernel.
- ⑤ A Linux process tries to create a new child process and calls the system-call `fork()`.
- ⑥ The calling process' (parent/current) `task_struct` is then passed to `sec_PVM`,
- ⑦ which requests the `KeyHMAC` from the `sec_Key_Storage` module and verifies the current process using this key.
- ⑧ Assuming that the current process was successfully verified by `sec_PVM`, the system-call `fork()` continues and copies the parent process. The child process' `task_struct` is then passed to `sec_PAM`.
- ⑨ `sec_PAM` also requests `KeyHMAC` from the `sec_Key_Storage` module and computes the HMAC of the child process `task_struct` using this key. Then it stores the computed HMAC into the child process `task_struct->ucaHMAC` field. The child process is then returned by `fork()` and ready to be executed.



- ⑩ Assuming the just created child process performs a different task than the parent process, it calls the system-call `exec()` to be prepared with the new executable code. The `exec()` system-call hooks a function which
  - ① loads the binary of the new code and computes the SHA256 value from this binary. The computed SHA256 value is then passed to the `sec_AVM` module to be verified.
  - ② The `sec_AVM` module requests the actual TAPL from the `sec_TAPL` module,
  - ③ and requests `Key_PubSig` from `sec_Key_Storage`, which is stored inside `sec_AVM`, to be used for further verifications. If the `Key_PubSig` has been already received from `sec_Key_Storage` this step will be skipped and leads directly to ④. Such approach lowers the communication overhead and accelerates the verification process for further verifications.
  - ④ The `sec_AVM` module requests the `sec_MTM` module to verify a given SHA256 value against the TAPL entries using the `Key_PubSig` one by one. This operation is repeated until one TAPL entry matches, or all TAPL entries have been processed without any match.
- ⑪ With a successful verification the system-call `exec()` continues and starts the new process continuing with the execution of the code from the binary file.
- ⑫ According to Section 7.3, the computed SHA256 value of the Android Application file is marshalled together with the pointer to a variable and passed to the `sec_Verify_Bridge` module (cf. Section 7.2.7). The attached SHA256 value is verified as described in the steps ① to ④. The result of this verification is then copied to the result variable (`cCheckPassed`) using the pointer from the marshalled request. With a successful verification, the Android Application is started inside the DVM.
- ⑬ An administrator requests an operation from the `sec_UI` module, as described in Section 6.1.5. The administrator is authenticated using the passed `Key_MTMAuth` from the marshalled request and
  - ⑭ if the authentication succeeds, `sec_UI` requests the corresponding operation from the `sec_TAPL` module.

### 7.4.2 Workflow for Android OS Layer

Performing a secure boot (cf. Section 3.1) passes the trust of the MTM to the verified bootloader, which then verifies the operating system and passes the trust received from the MTM to it. Figure 7.3 illustrates the Android OS boot process with the implemented security architecture until

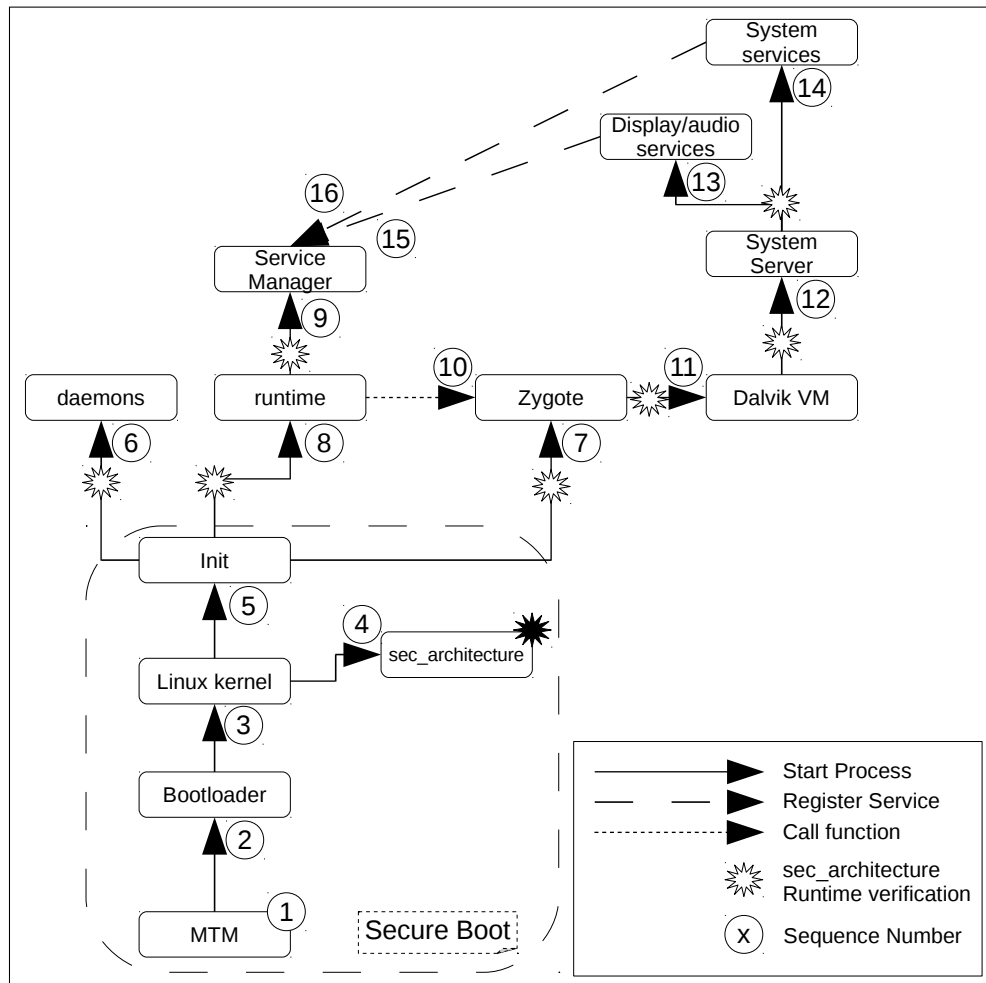




Figure 7.3: Android OS boot sequence with security architecture

the first Android Application can be started. The  represents the points where a runtime verification is performed by the security architecture. The security architecture modules are commulated displayed and marked with  as one sec\_architecture entity. It is assumed that all kernel modules from the implemented security architecture are loaded successfully at this point. The bulletpoints ① to ⑯ describes the performed step(s) in the associated point depicted in Figure 7.3.

- ① The MTM perform a self measurement and integrity check (cf. Figure 3.1 ①).
- ② After a succesful verification of the device integrity including the MTM and the bootloader, the bootloader is executed (cf. Figure 3.1 ⑤).

- ③ The bootloader is designated to calculate the SHA256 from the operating system binaries and let them verify against values taken and stored by the MTM in a secure and trusted state. Then, the bootloader executes the kernel which starts and prepares further operations.
- ④ The kernel loads drivers, kernel modules and the modules from the security architecture.
- ⑤ Subsequently, the INIT process is created. At the end of the MTM's secure boot (cf. Figure 3.1 ⑧). The involved secure boot parts are accented with a surrounding fine dashed rectangle.
- ⑥ Before starting any daemon or service, which implies a new linux process start, the binaries are verified by the `sec_AVM` module (cf. Section 7.2.3).
- ⑦ - ⑩ All processes created and executed after ⑥, until the first Android Application can be executed, are catch by the security architecture and verified using the `sec_AVM` module.

After the step ⑩, the device is in a verified and a secure state. It can now perform further runtime integrity verifications using the security architecture. Figure 7.4 illustrates the interaction between the Android OS DVM and the security architecture modules performing a runtime integrity verification. It shows the Activity Manager as step ① which is an Android OS service that initiates new Android Activities (new Android Application starts). The Package Manager is an Android OS service which manages all installed and available Android Applications on the mobile device. The bulletpoints ① to ⑧ shows the workflow starting one Android Application and the interaction with the security architecture in user-space. The further bulletpoints ① to ⑤ show the verification sequence of an Android Application accessing the security architecture in kernel-space by using the `/dev/sec_device_vb` file and the `sec_Verify_Bridge` module.

The first part covers the user-space part of an Android Application start.

- ① A request to start a new Android Application is sent to the Activity Manager.
- ② It sends a request to Zygote to `fork()` a new DVM.
- ③ Zygote sends a request to the kernel to `fork()` which starts the HMAC verification as depicted in Figure 7.1.
- ④ The kernel `fork()`s a new DVM.
- ⑤ The Package Manager passes information about the new Android Application to the DVM.
- ⑥ The DVM hooks a verification function which loads the Android Application file and computes a SHA256 value of it.

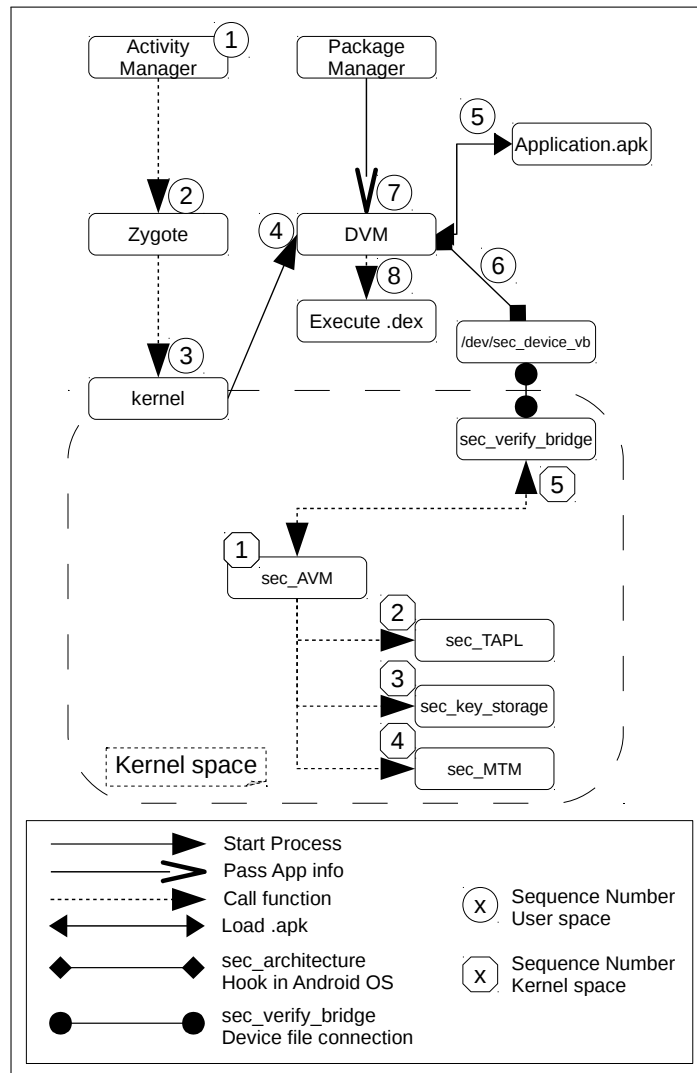


Figure 7.4: Android OS application start with security architecture

- ⑦ The SHA256 value is marshalled and written to the `/dev/sec_device_vb` file according to Section 7.2.7. This initiates a verification process using the passed marshalled data in kernel-space.
- ⑧ If the hooked verification function returns a success code the `.dex` file is extracted from the file and executed in the DVM. Otherwise an exception is thrown to the Android OS.

The following part covers the kernel-space verification of an Android application start.

- ⑤ The `sec_verify_bridge` receives the written data from ⑦ and unmarshals them. This point is the entry and exit point to communicate through the `/dev/sec_device_vb` device file. It locks the function and disables all IRQs.
- ① The `sec_verify_bridge` calls the `sec_AVM` to verify the SHA256 passed from the DVM.
- ② The `sec_AVM` requests the `sec_TAPL` module to receive the TAPL.
- ③ Then it requests then the `KeyPubSig` from `sec_key_storage`
- ④ and delegates the implemented MTM stub (`sec_MTM`) to verify the SHA256 using the `KeyPubSig` against all TAPL entries.
- ⑤ The verification result is copied to the result variable passed together with the marshalled data from user-space. Then the IRQs are restored and the function is unlocked.

## 8 Evaluation

This section discusses the assumptions made for this thesis at first and the actual results grown while researching and implementing the discussed security architecture. Later a conclusion relying on the elaborated results is given and finally an outlook for further development and improvement of this security architecture is made.

### 8.1 Development Assumptions

#### 8.1.1 General Assumptions

The proposed security architecture was assumed for mobile devices running on a linux kernel and to be protected primary by an MTM hardware module. It was well known at the beginning of this thesis, that no MTM hardware is currently available. TPM modules are already used in currently available Laptops it is expected that MTM modules for mobile devices be available in the near future. Therefore, the proposed security architecture was developed while relying on an MTM hardware. Microsoft<sup>1</sup>'s BitLocker Drive Encryption uses such a TPM module to secure data on a harddisk when lost or stolen [Ltd10].

The specification of the MTM meets the requirements for the proposed security architecture. However, comparable hardware is already available, such as the ARM TrustZone. The ARM TrustZone is a feature for special ARM family CPUs, e.g. the ARMv6KZ. It allows to separate instructions from trusted and untrusted programs using a hardware based virtualisation and separation of the CPU and the stack for trusted and untrusted programs, which have to be classified before execution. This virtualisation separates both environments from each other (trusted and untrusted) such that they cannot physically interfere with each other. ARM TrustZone provides immutable memory to store sensible information such as keys or other confidential data. In contrast to the MTM, it is not intended for remote attestation [Ltd11].

---

<sup>1</sup>microsoft is an operating system and software development company.

## Android OS

Because Android OS is running on a linux kernel and is a free and open source operating system, the proposed security architecture is implemented on the Android OS. Furthermore, it is an established system used by a widely range of mobile devices. Difficulties mainly arose while analysing the Android OS and especially the Virtual Machine model used for Android application execution. Linux programs were catch at the `exec()` call and verified by the proposed security architecture. Android Applications are executed inside of a DVM, which is `fork()`ed from Zygote, but not `exec()`ed by the kernel. The `fork()`ed DVM acts in the same way as Zygote, thus it does not need to be `exec()`ed to to execute a code. The DVM bytecode provided by Android Applications is directly interpreted inside of the DVM. This situation resulted adopting the proposed security architecture for the the Android OS. As debugging of a whole operating system is a complex task, the actual point (`androidOS/dalvik/vm/native/dalvik_system_DexFile.c`) with the placed hook to connect with `/dev/sec_device_vb` was found mainly by reading and analysing the source code of Android OS. However, it may be possible that there are alternative positions Assuming that, there is possibly in the Android OS to hook into the security architecture verification. The used position of the hook has proven that it is at least one of the points applicable for runtime verification of the proposed and implemented security architecture.

### 8.1.2 Kernel Modifications

Data is read directly from the filesystem of the kernel-space by the security architecture. This includes the `sec_TAPL.txt` file and the binary data of the processes calling the `exec()` system-call, to be used in computing the SHA256 value of this binary. This behaviour is in conflict with the principle, that kernel-space tasks should not perform user-space tasks and vice versa. It is noted, that such tasks should be performed by a daemon providing the named abilities. Anyhow, such daemon would have to be started directly after the INIT process and before any other processes. Additionally, it would have to be assured that such daemon is not replaced or bypassed during runtime. Directly accessing and reading files from within the kernel-space eases these problems.

### 8.1.3 PAM and PVM

The authentication of processes with HMAC was proposed a an additional security feature for execution control at runtime. This allows that only processes verified by the PVM are allowed to `fork()` and to create child processes, that are in turn authenticated by the PAM. This process verification with PVM is also hooked into the scheduler. It was intended to prevent unverified

processes from being scheduled and to kill them if not trusted. Tests were performed during development using the implemented security architecture. The `sec_AVM` verification caught all untrusted programs and Android Applications during their creation. Therefore, there were not untrusted processes running and trying to `fork()` or to be scheduled. The authentication and verification of “already” trusted processes with PAM and PVM increase the total security overhead. However, this approach can be useful in scenarios where the processes are modified during runtime. Moreover, the hooked points for PAM and PVM are possibly usable to perform other checks on the `task_structs`, e.g. as a runtime code section analysis.

### 8.1.4 Cryptographic Abilities

At the beginning of the implementation phase the security architecture used stub functions for cryptographic calculations such as the signature verification, the SHA256 and the HMAC computations. An attempt was made to use the `OpenSSL` library for this task. This library provides all kind of hash functions. Unfortunately, this library cannot be used with the security architecture, as it tries to obtain memory in a way that locks the whole kernel. Additionally, the `OpenSSL` library does not provide any asymmetric cryptography functions, which would be necessary to fully simulate the cryptographic abilities of the MTM. Therefore, some modified parts of the `OpenSSL` library are used in the security architecture to compute SHA256 values. The signature verification (asymmetric cryptography) is still implemented as a stub function. It extends a given SHA256 from 256 bits to 4096 bits and pad the additional bits with 0.

## 8.2 Development Platform and Prototyping

The development platform is the OMAP4430 Panda Board ES2.1 type(GP). This platform is equipped with a 1.5 GHz dual core ARM Cortex-A9 processor and with 1GB Low Power DDR2 RAM (cf. Figure 8.1). The operating system is compiled from the source (Android OS Gingerbread) with all patches applied for the OMAP4430 Panda Board. The kernel is (Android linux kernel 2.6.35) applied with all patches according to [OMA11].

### 8.2.1 Command Line Configuration Tool - `sec_configure` Tool

The command line configuration tool `sec_configure` tool was developed to perform administrative tasks on the security architecture. Once loaded the user is welcomed with the main opening screen as depicted in Figure 8.2.



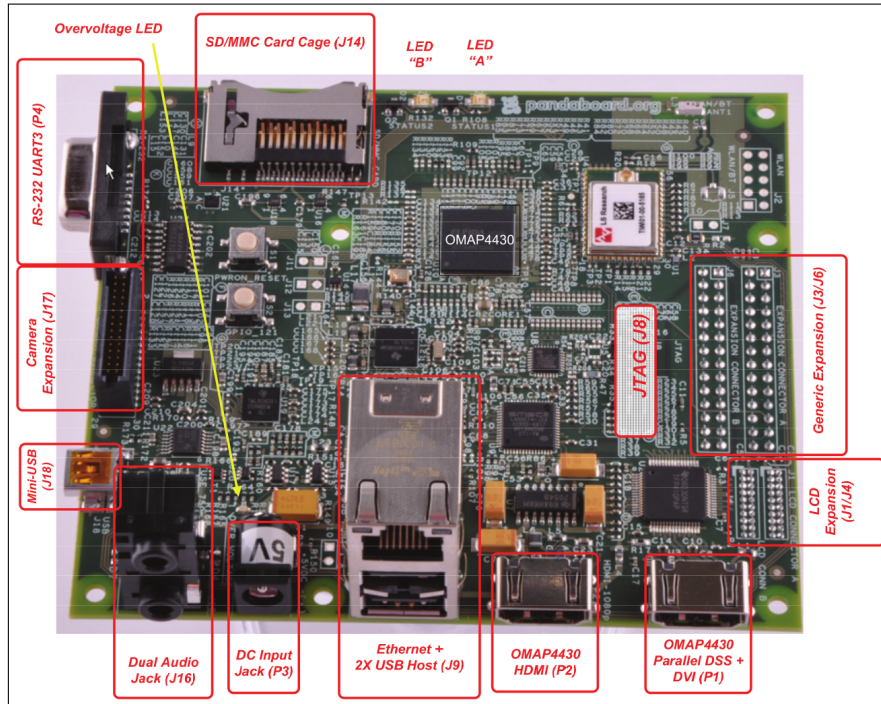


Figure 8.1: The OMAP4430 Panda Board Evaluation Platform (cf. [Pan])

## Main Menu

It provides different options to the administrator. The most important options are:

[r] this option reads the `sha256_entries.txt` file into a buffer. This file contains the list of all SHA256 values from all executable files of the OS. The entries of this file are plain text entries where each line contains the SHA256 stored in hexadecimal format and the binary name separated by a white-space character, e.g.

```
90650013420cc42ae250a509d8c8e5f5d693447ebfa94aa5becee0ae9b5c5805 sec_configure
```

[m] this option reads the `sha256_minimal_entries.txt` into a buffer. This file contains a subset of entries of the file used with [r], composed of hand-picked entries to improve the readability.

[p] this option is used to go through a loaded (buffered) list and request the security architecture to add each entry of the list to the TAPL one by one. The requests are marshalled according to Section 7.2.6 and then passed to the `/dev/sec_device` using the `ioctl(...)` system-call. The administrator is prompted to enter the authentication key `KeyMTMAuth` to be authenticated by the MTM if not done before using the option [z].

```

#####
#                               -= sec_configure Tool v.0.2 alpha -=                               #
#                               ::::::::::::::::::::                               #
#                               #####                               #
#                               ::::::::::::::::::::                               #
#                               #####                               #
#                               ::::::::::::::::::::                               #
#                               #####                               #
#                               ::::::::::::::::::::                               #
#                               #####                               #
#####
#####
#                               Main Menu                               #
#                               =====                               #
#                               #                               #
# [r] read sha256 file into buffer                                     #
# [m] read minimal sha256 file into buffer                           #
# [p] add all sha256 from buffer to kernel TAPL                     #
# [k] read TAPL from kernel and print it (for debug)                #
# [l] list all read sha256 entries (whole)                           #
# [n] list all read sha256 entries (names only)                     #
# [x] Remove Menu (remove one entry from kernel TAPL)              #
# [a] Add Menu (add one entry to kernel TAPL)                       #
# [z] enter Password                                                #
#                               #                               #
# [q] to quit tool                                                  #
#####
:

```

Figure 8.2: Security Architecture Command Line Configuration Tool - Main Menu

- [x] this option displays a new menu providing options to remove entries from the security architecture's TAPL
- [a] this option displays a new menu providing options to add single entries to the security architecture's TAPL
- [z] this option prompts the administrator to enter the authentication key  $Key_{MTMAuth}$  which is saved to be used further in this session
- [q] quits the sec\_configure Tool

### Remove Menu

- [n] this option lists all entries from the actual loaded (buffered) list
- [b] this option is used to return to the Main Menu
- [x] this option prompts the administrator to enter the index of the entry which will be removed from the TAPL

```

#####
#                                     #
#                               Remove Menu                               #
#                               =====                               #
#                                     #
#   [n] list all read sha256 entries (names only)                       #
#   [b] back to main menu                                               #
#   [x] remove one entry from kernel TAPL                               #
#                                     #
#####
:
    
```

Figure 8.3: Security Architecture Command Line Configuration Tool - Remove Menu

**Add Menu**

```

#####
#                                     #
#                               Add Menu                               #
#                               =====                               #
#                                     #
#   [n] list all read sha256 entries (names only)                       #
#   [b] back to main menu                                               #
#   [a] add one entry to kernel TAPL                                    #
#                                     #
#####
:
    
```

Figure 8.4: Security Architecture Command Line Configuration Tool - Add Menu

The [n] and [b] options as in the Remove Menu.

[a] this option prompts the administrator to enter the index of an entry to add into the TAPL. If  $Key_{MTMAuth}$  is not entered before, the administrator is prompted to enter it. The administrator is informed whether the operation has succeeded with messages “adding sec\_configure to TAPL succeed!” and “adding sec\_configure to TAPL failed!”.

**8.2.2 GUI for the configuration Tool - sec\_Configuration Tool**

The sec\_Configuration Tool was implemented to make administration of the security architecture more comfortable. It uses the same native implementation for accessing the security architecture as the command line tool. In contrary to it, the GUI manages the security architecture using the JAVA environment of Android OS and its native interface.

When the GUI is started, the administrator is welcomed with the main window as depicted in Figure 8.5. In the upper-left part of the main window a password field is displayed (●●●●●●). The administrator enters the  $Key_{MTMAuth}$  here to be authenticated if requesting a signing operation from the MTM. At the bottom-left of the main window, the type of the list to be displayed

can be chosen by setting one of the radiobuttons  (SHA256 List) or  (TAPL) to open a corresponding view. The buttons below the radiobuttons are used to remove/add entries from or to the TAPL.

**View - SHA256 List**

The SHA256 List view is used to display the list of SHA256 entries. One can request the security architecture to sign selected entries from this list and add them to the TAPL. Activating the  (SHA256 List) radiobutton calls a fuction to read the sha256\_entries.txt file from the filesystem. The read in text is then passed to the Android OS JAVA interface, where the entries are parsed and prepared to be shown GUI as illustrated in Figure 8.5. This is similar to the [r] option provided by the command line Main Menu. The  Filter (.apk) checkbox can be used to display a subset of .apk files.

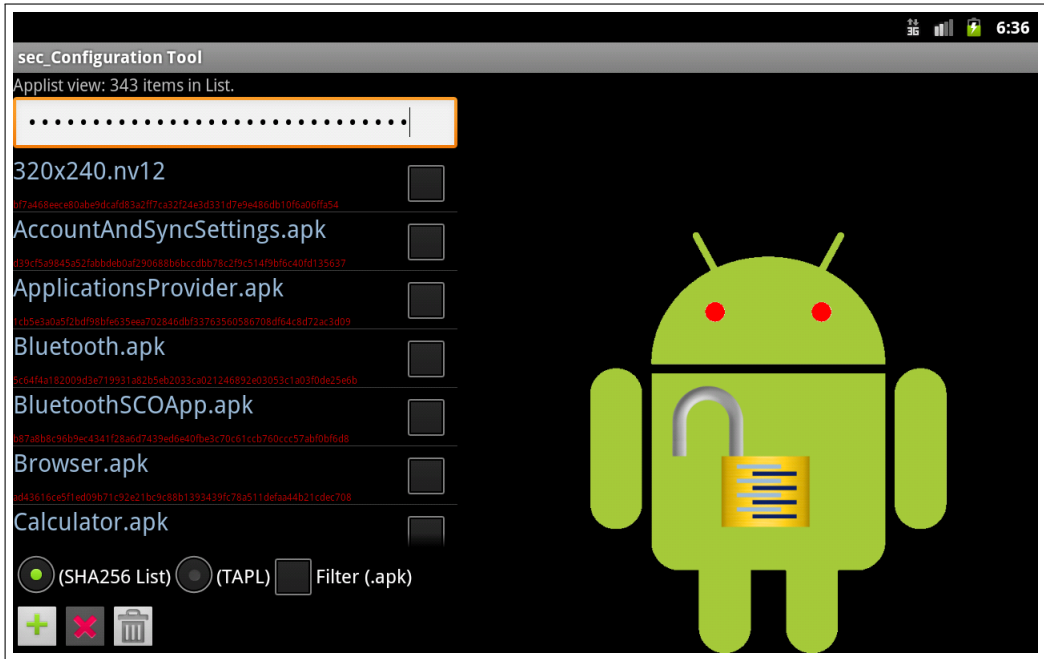


Figure 8.5: GUI - List showing the SHA256 of each program available on the device



Figure 8.6: The List-entry for the Calculator Application

Each entry in the list contains the name of the program and below of the name the corresponding SHA256 is displayed in red. This is illustrated in Figure 8.6 for the Calculator.apk entry as an example. On the right side of each entry a checkbox is displayed which can be checked to select the entry ()

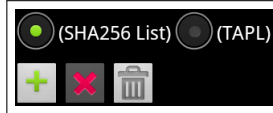


Figure 8.7: The control boxes available by selecting the  (SHA256 List)

As illustrated in Figure 8.7 in this view the  button is activated. Pressing this button requests the configuration Tool to collect all of the entries marked in the list and the  $Key_{MTMAuth}$  from the password field. The marked entries and the  $Key_{MTMAuth}$  are passed to the underlying native function requesting the security architecture to sign and add these entries to the TAPL one by one. Figure 8.8 shows three entries marked to be signed and added to the TAPL. If the  $Key_{MTMAuth}$

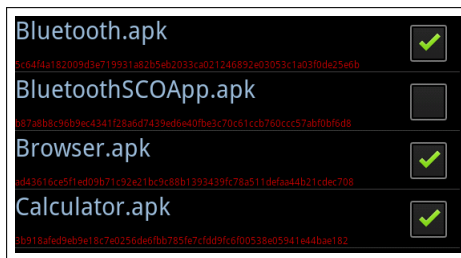


Figure 8.8: Marked entries in the SHA256 list

is already entered, pressing the  button will add the entries to the TAPL. The administrator will be informed about the result of the request as depicted in Figure 8.9 and Figure 8.10. If no password is set or no list entries are marked the administrator is also informed about this situation. The garbage button depicted in Figure 8.7 and 8.13 (rightmost button on the bottom) un-checks all checked entries in the actually used list.

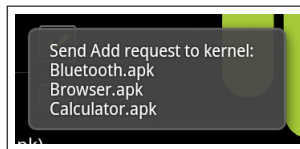


Figure 8.9: Request performed successfully



Figure 8.10: Request not performed due to authorisation failure

### 8.2.3 View - TAPL

The TAPL view is used to show the actual TAPL and to request to remove entries from it. Once activated, it calls the native function to request the actual TAPL form the security architecture. When received, it is passed from the native function to the Android OS JAVA interface, to be shown in the GUI as illustrated in Figure 8.11. The list shows the signed entries stored in the TAPL (sec\_TAPL) TAPL. The  Filter (.apk) checkbox can be checked to hide all non .apk files in the displayed list. Each entry in the list shows the name of the program and below the name,

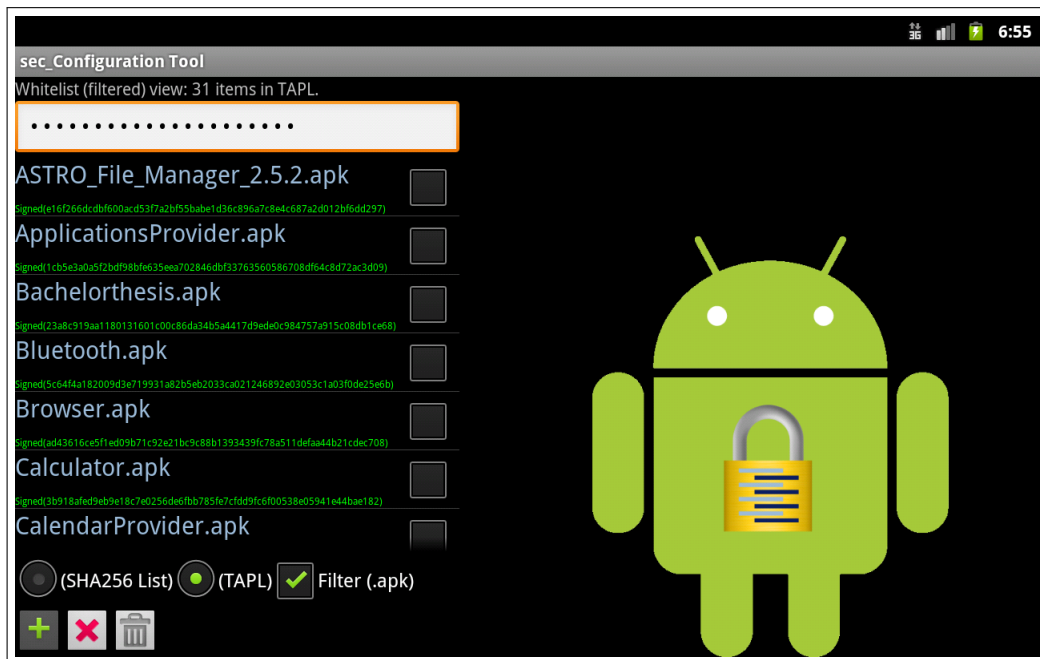
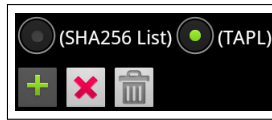



Figure 8.11: The TAPL view showing the list of all signed programs

the corresponding signed SHA256 value is displayed in green. This is illustrated in Figure 8.12 for the Calculator.apk as an example. Corresponding to Section 8.2.2, these entries can also be marked by activating the checkbox and unchecked all together by using the garbage button. In this view, the  button is activated as illustrated in Figure 8.13. Pressing this button requests



Figure 8.12: The signed List-entry for the Calculator Application

Figure 8.13: The controls boxes available by activating the  (TAPL)

the configuration Tool to collect all list entries and to request the security architecture to remove the entries from the TAPL one by one.

#### 8.2.4 Automated Build Script

During implementation of the security architecture, a bash script was developed. This script was written to ease the build process and to make it configurable for future builds.

It comes along with a bundle of .diff files in addition to the security architecture source files. Subfolders required during a build are generated automatically by the script. The script can be configured by changing variables at the very beginning of it. Logfiles are generated and saved in a separate subfolder for debugging purposes.

This script automates:

- fetching the compiler and third party drivers (e.g. WLAN and Bluetooth drivers) for the OMAP4430 Panda Board
- fetching the sources for Android OS and the kernel for the OMAP4430 Panda Board
- fetching the bootloader
- fetching and applying the patches for the bootloader, e.g. to enable booting directly from SDCard without the need of typing boot-options in the command line
- fetching the patches for the kernel and applying them
- fetching the patches for the Android OS and applying them
- applying additional OMAP4430 Panda Board changes/bugfixes not available with patches
- applying the security architecture patches, enabling the security architecture in the following compiled build

- compiling the bootloader
- compiling third party drivers
- compiling the kernel
- compiling the security architecture modules
- compiling the command line security architecture configuration tool
- compiling the Android OS
- collecting the right data and configuring the Android OS, e.g. edit the `init.rc` to automatically load the security architecture modules
- computing the SHA256 of all executable binaries and .apk files and storing them to the `sha256_entries.txt`
- generating an initial TAPL to enable at least to boot into Android OS, including the configuration tools for the security architecture
- preparing the folderstructure to be used with the SDCard and copying the data to it

### 8.3 Performance Evaluation

Measuring the performance overhead caused by the security architecture during boot has been measured on the OMAP4430 Panda Board. The operating system and all programs/Applications are loaded from an SDCard<sup>2</sup> of Class 6<sup>3</sup>. The Android OS resides on an ext3<sup>4</sup> type partition on the SDCard. Contrary a mobile device has to be flashed with the Android OS using an image file (flashed image). The kernel and the boot loader are stored on a FAT32<sup>5</sup> type partition.

The time measurements were performed using the kernel's high resolution timer defined in `<linux/hrtimer.h>`. The measurements were taken closely around the hooks from the security architecture. That are:

- inside the `exec()` system-call measuring the time consumed by reading a file, computing the SHA256 value and verifying it using the `sec_AVM` module.

---

<sup>2</sup>SDCard (Secure Digital Memory Card) a memory chip stored on a removeable card to flash data to it and to read the data from the card.

<sup>3</sup>this classification informs about the minimal transfer rate when writing data to the card

<sup>4</sup>ext3 (third extended filesystem) a filesystem architecture that monitors changes done to files, which ensures consistency if a writing operation was interrupted by e.g. a systemcrash.

<sup>5</sup>FAT32 (File Allocation Table) a filesystem architecture.



Test	Size in bytes	Time
1	64	0 $\mu s$
2	128	0 $\mu s$
3	1024	30 $\mu s$
4	102400 (100KB)	3143 $\mu s$
5	102400 (100KB)	3144 $\mu s$
6	102400 (100KB)	3143 $\mu s$
7	1024000 (1000KB)	33234 $\mu s$
8	1024000 (1000KB)	32807 $\mu s$
9	1024000 (1000KB)	32288 $\mu s$
$\Sigma$ w/o 1+2	3380224	78789 $\mu s$

Table 8.1: Time consumptions for computing SHA256 values

- while calling the sec\_AVM module to verify a SHA256 value.
- while calling the sec\_Verify\_Bridge module from the DVM, to verify an Android Application.

The resolution for all measurements was  $\mu sec$ . A kernel module (sec\_timeTetsts.ko) has been used to perform reference measurements for computing SHA256s values. The SHA256 values are computed from vmallocated heap. Loading this module into the kernel cause ot to perform 9 measurements. The measurements performed over data scaling from 64bytes to approximately 1MB an nine measurements are performed. The results of these reference measurements are listed in Table 8.1. They indicate a performance overhead of  $0,023 \frac{\mu s}{byte}$  by computing a SHA256 value, not including the first two measurements of Table 8.1. Table 8.2 shows the measured times during the boot and at the exec () system-call. Table 8.3 lists the measurements for verifying Android Applications from within a DVM, using the sec\_Verify\_Bridge module. The measured times listed in 1 and 2 had a duration  $< \mu s$ , so they are not considered in the foolowing calculations. The total time  $\Delta$ , produced by the verification through the security architecture during boot, can be calculated using the measured times from Table 8.2 and 8.3. Native linux processes consume 481203 $\mu s$  for a total of 825892bytes. The verification of their SHA256 values consume 3725 $\mu s$ . The required time to read a byte from filesystem and to compute a SHA256 over it is therefore approximately

$$\frac{481203\mu s - 3725\mu s}{825892bytes} = 0,58 \frac{\mu s}{byte} \quad (8.1)$$

Compared to the result calculated from the time measurement test from Table 8.1 ( $0,023 \frac{\mu s}{byte}$ ), reading a byte of data from the filesystem takes  $\approx 24$  times larger than the time to compute a SHA256 of one byte data. The time calculated in Equation 8.1 is used to calculate the time consumed by the verification of Android Applications. There are 25 Android Applications loaded during the boot, with a total size of 14127921bytes. The verification process of these Applications took 10416 $\mu s$ .

Name	Size in bytes	Time:	
		• reading file • computing SHA256 • verification sec_AVM	• verification sec_AVM
ueventd	90084	5951 $\mu$ s	122 $\mu$ s
servicemanager	9940	3052 $\mu$ s	214 $\mu$ s
mediaserver	5488	2075 $\mu$ s	92 $\mu$ s
rild	9808	8697 $\mu$ s	122 $\mu$ s
app_process	5720	2319 $\mu$ s	122 $\mu$ s
vold	51700	19684 $\mu$ s	122 $\mu$ s
netd	41636	14069 $\mu$ s	122 $\mu$ s
dbus-daemon	22420	20721 $\mu$ s	91 $\mu$ s
installd	109504	30854 $\mu$ s	122 $\mu$ s
keystore	10112	8698 $\mu$ s	92 $\mu$ s
debuggerd	18112	18249 $\mu$ s	122 $\mu$ s
pvrsvinit	5412	13428 $\mu$ s	183 $\mu$ s
insmod	81544	36224 $\mu$ s	153 $\mu$ s
uim-sysfs	13792	3754 $\mu$ s	122 $\mu$ s
adbd	113876	71320 $\mu$ s	183 $\mu$ s
bootanimation	23160	7233 $\mu$ s	214 $\mu$ s
wlan_loader	9716	140259 $\mu$ s	519 $\mu$ s
tc	67956	65308 $\mu$ s	214 $\mu$ s
tc	67956	4516 $\mu$ s	519 $\mu$ s
tc	67956	4792 $\mu$ s	275 $\mu$ s
$\Sigma$	825892	481203 $\mu$ s	3725 $\mu$ s

Table 8.2: Performance overhead during the boot (Linux Processes)

That is,

for the SHA256 computation for the Android Applications:

$$14127921\text{byte} \cdot 0,58 \frac{\mu\text{s}}{\text{byte}} = 8194194,18\mu\text{s} \quad (8.2)$$

together with the time required for the verification of these files:

$$8194194,18\mu\text{s} + 10416\mu\text{s} = 8204610,18\mu\text{s} \quad (8.3)$$

which results, together with the time for verifying the linux native proram, in a total of:

$$8204610,18\mu\text{s} + 481203\mu\text{s} = 8685813,18\mu\text{s} \quad (8.4)$$

The absolute performance penalty is  $\Delta \approx 8,7\text{s}$ , compared to an Android OS boot sequence without the implemented security architecture. According to the result, the integrity measurement for the operating system's core libraries ( $\approx 24,7\text{MB}$ ) would take about  $\approx 15\text{s}$ . This would produce a total penalty of  $\approx 23,7\text{s}$  at startup, e.g. during a secure boot (cf. Figure 3.1 ⑤ to ⑧).

Name	Size in bytes	Time:	
		• verification sec_AVM • copy from and to DVM	• verification sec_AVM
SettingsProvider.apk	45626	502 $\mu$ s	488 $\mu$ s
SystemUI.apk	225334	306 $\mu$ s	245 $\mu$ s
LatinIME.apk	773192	275 $\mu$ s	275 $\mu$ s
Launcher2.apk	841334	305 $\mu$ s	152 $\mu$ s
Phone.apk	2366837	336 $\mu$ s	275 $\mu$ s
TelephonyProvider.apk	59828	335 $\mu$ s	274 $\mu$ s
UserDictionaryProvider.apk	11119	335 $\mu$ s	305 $\mu$ s
ContactsProvider.apk	164367	824 $\mu$ s	275 $\mu$ s
ApplicationsProvider.apk	24077	305 $\mu$ s	275 $\mu$ s
DownloadProvider.apk	418969	366 $\mu$ s	305 $\mu$ s
MediaProvider.apk	55133	397 $\mu$ s	275 $\mu$ s
DrmProvider.apk	26655	305 $\mu$ s	305 $\mu$ s
Mms.apk	1188735	305 $\mu$ s	336 $\mu$ s
DeskClock.apk	373311	336 $\mu$ s	274 $\mu$ s
javax.obex.jar	26527	336 $\mu$ s	275 $\mu$ s
Bluetooth.apk	384212	244 $\mu$ s	214 $\mu$ s
VoiceDialer.apk	113824	306 $\mu$ s	274 $\mu$ s
android.test.runner.jar	76251	458 $\mu$ s	305 $\mu$ s
CalendarProvider.apk	367248	244 $\mu$ s	214 $\mu$ s
Email.apk	1212554	336 $\mu$ s	275 $\mu$ s
Music.apk	680915	367 $\mu$ s	336 $\mu$ s
Protips.apk	120068	336 $\mu$ s	305 $\mu$ s
QuickSearchBox.apk	569446	610 $\mu$ s	550 $\mu$ s
Gallery3D.apk	635488	641 $\mu$ s	275 $\mu$ s
Settings.apk	3978871	604 $\mu$ s	549 $\mu$ s
$\Sigma$	14127921	10416 $\mu$ s	7631 $\mu$ s

Table 8.3: Performance overhead during the boot (Android Applications)

## 8.4 Conclusion

The proposed security architecture is capable to detect code manipulations after the installation of the code. A hash value (SHA256) from a modified binary always differs the hash value computed from the same but unmodified binary. Signing the hashes using a secure signature algorithm protects them against modifications. Accomplishing this, manipulated binaries are always successfully detected and prevented from execution. Native linux programs are verified by the security architecture when calling the `exec()` system-call (cf. Section 7.4 (10) et seq.). Android Applications are verified using the inserted security hook inside the DVM (cf. Section 7.4 (12) et seq.). Both verification points cover all program starts on the Android OS and protect it to be harmed. Compared to anti-malware program, which analyses the behaviour of a process at runtime, such a signature based approach is low-cost in terms of computation and power consumption. As it is a white-list approach it has to verify a program against allowed entries.

This is usually a subset of all programs runnable on the mobile device. The security architecture is not able to verify any runtime loaded library or code from actual executing code, which is out of the scope of this security architecture. Opposed to that, the security architecture can detect any manipulation on the verified binaries and prevent executing them without having knowledge about the kind of manipulation and a possible malicious behavior. This protects the mobile device from even zero-day malwares which are not detectable by anti-malware programs, because unknown until then. Assuming a mounted MTM hardware module, a mobile device is able to be remotely attested and to perform runtime verification of executed program binaries using the implemented security architecture. Moreover, the mobile device through the MTM, is able to perform a secure boot providing the trust anchor for the security architecture. An MTM hardware discharges the CPU to verify SHA256s from the TAPL using the  $\text{Key}_{\text{PubSig}}$ . It also stores and protects the used keys in shielded locations, signs given SHA256s and can authenticate administrators. It is assumed that the costs for computation tasks are less as if performed by software components of the operating system. This includes the overhead caused by communicating with an MTM hardware module (cf. Section 8.3). The implementation of the security architecture has shown, that the primary elaborated goals of runtime verification and manipulation detection have been reached. The performance overhead of the proposed security architecture is also acceptable. Even without an MTM module mounted on the mobile device, using the proposed security architecture grant an administrator to have full control of which programmes are executable on it.

### 8.5 Outlook

The kernel modules can be protected against reloading or be fully integrated into the kernel.

The communication between the administrator and the security architecture can be protected through a daemon to disable the possibility to intercept messages passed to the kernel especially the  $\text{Key}_{\text{MTMAuth}}$ .

The security architecture can possibly be “lifted” up into a higher layer without losing its atomic connection to the system. This could make the security architecture more lightweight.

The initial trust anchor provided by MTM could be replaced by a software driven trust anchor providing the same or an acceptable level of trust which the security architecture could derive from.

A dynamic loaded code verification at runtime similar to the performed verification could be implemented additionally.

Recycling the PAM and PVM to extend the existing security architecture to stop a process e.g. reading manipulated libraries cleanly at scheduling.

## Bibliography

- [Bor08] Dan Bornstein. *Dalvik VM Internals*. Google I/O Webpage/Video. Nov. 2008. URL: <http://sites.google.com/site/io/dalvik-vm-internals> (visited on 11/22/2011).
- [Bra08a] Patrick Brady. *Anatomy & Physiology of an Android*. Google I/O Webpage/Video. Nov. 2008. URL: <http://sites.google.com/site/io/anatomy--physiology-of-an-android> (visited on 11/22/2011).
- [Bra08b] Patrick Brady. *Anatomy & Physiology of an Android*. Google I/O Webpage/Slides. Nov. 2008. URL: <http://sites.google.com/site/io/anatomy--physiology-of-an-android/Android-Anatomy-GoogleIO.pdf?attredirects=0> (visited on 11/22/2011).
- [EK11a] Elektronik-Kompendium.de. *Prozessor (CPU)*. Webpage. 2011. URL: <http://www.elektronik-kompendium.de/sites/com/0309161.htm> (visited on 11/23/2011).
- [EK11b] Elektronik-Kompendium.de. *RAM - Random Access Memory*. Webpage. 2011. URL: <http://www.elektronik-kompendium.de/sites/com/0309191.htm> (visited on 11/23/2011).
- [FSF11] Inc. Free Software Foundation. *GNU General Public License*. Website. Sept. 2011. URL: [http://en.wikipedia.org/w/index.php?title=Processor\\_register&oldid=455430304](http://en.wikipedia.org/w/index.php?title=Processor_register&oldid=455430304) (visited on 11/23/2011).
- [Goo11a] Google. *API*. Google API Webpage. Nov. 2011. URL: <http://developer.android.com/reference/packages.html> (visited on 11/22/2011).
- [Goo11b] Google. *Google history*. Webpage. Oct. 2011. URL: <http://www.google.com/intl/en/about/corporate/company/history.html> (visited on 11/23/2011).
- [Goo11c] Google. *What is Android?* Android Developers Webpage. Nov. 2011. URL: <http://developer.android.com/guide/basics/what-is-android.html> (visited on 11/22/2011).

- [KKP09] Prof. Dr. Bernd Kahlbrandt, Prof. Dr. Christoph Klauck, and Prof. Dr. Stephan Pareigis. *Algorithmen und Datenstrukturen für Angewandte und Technische Informatiker*. Lecture notes from winter term 2010/11. Berliner Tor 7, 20099 Hamburg: Department für Informatik - Hochschule für Angewandte Wissenschaften Hamburg, 2009.
- [Lov05] Robert Love. *Linux-Kernel-Handbuch*. Addison-Wesley Verlag, 2005. ISBN: 3-8273-2247-2.
- [Ltd10] Microsoft Ltd. *BitLocker Drive Encryption*. Webpage. May 2010. URL: <http://technet.microsoft.com/en-gb/library/cc731549%28WS.10%29.aspx> (visited on 11/23/2011).
- [Ltd11] ARM Ltd. *TrustZone*. Webpage. Nov. 2011. URL: <http://www.arm.com/products/processors/technologies/trustzone.php> (visited on 11/23/2011).
- [MOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. 5th ed. Corporate Blvd., Boca Raton, Florida 33431: CRC Press, 2001. ISBN: 0-8493-8523-7.
- [Mtm] *TCG Mobile Trusted Module Specification*. Revision 7.02. Apr. 2010.
- [Nau+10] Mohammad Nauman et al. "Beyond Kernel-level Integrity Measurement: Enabling Remote Attestation for the Android Platform". In: *Trust and Trustworthy Computing (TRUST)* (2010).
- [OMA11] OMApedia. *PandaBoard L27.12.1-P2 Release Notes*. Webpage. 2011. URL: [http://www.omappedia.com/index.php?title=PandaBoard\\_L27.12.1-P2\\_Release\\_Notes&oldid=14324](http://www.omappedia.com/index.php?title=PandaBoard_L27.12.1-P2_Release_Notes&oldid=14324) (visited on 11/23/2011).
- [Pan] *OMAP 4 PandaBoard System Reference Manual*. Revision 0.4. Sept. 2010. URL: <http://omappedia.org/index.php?title=PandaBoard&oldid=13742> (visited on 11/22/2011).
- [Sai+04] R. Sailer et al. "Design and implementation of a TCG-based integrity measurement architecture". In: *the 13th conference on USENIX Security Symposium (SSYM'04)* (Aug. 2004).
- [SBP07] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. *The Linux Kernel Module Programming Guide*. Webpage. May 2007. URL: <http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html> (visited on 11/22/2011).
- [Tpm] *TPM Main Part 1 Design Principles*. Specification Version 1.2 Level 2 Revision 103. June 2007.

- [UIF11] Inc. USB Implementers Forum. *RAM - Random Access Memory*. Webpage. 2011. URL: <http://www.usb.org/home> (visited on 11/23/2011).
- [UW11] Osman Ugus and Dirk Westhoff. "An MTM based Watchdog for Malware Famishment in Smartphones". In: *I2CS 2011, Berlin, Germany* (June 2011).
- [UWL11] Osman Ugus, Dirk Westhoff, and Martin Landsmann. "A Multilevel Security Architecture for App Signature Verification and Process Authentication for Smartphones". In: *not released* (2011).
- [Wika] *Booting*. Wikipedia. Nov. 2011. URL: <http://en.wikipedia.org/w/index.php?title=Booting&oldid=461672254> (visited on 11/23/2011).
- [Wikb] *Firmware*. Wikipedia. Nov. 2011. URL: <http://en.wikipedia.org/w/index.php?title=Firmware&oldid=461089631> (visited on 11/23/2011).
- [Wikc] *Flash memory*. Wikipedia. Nov. 2011. URL: [http://en.wikipedia.org/w/index.php?title=Flash\\_memory&oldid=461609995](http://en.wikipedia.org/w/index.php?title=Flash_memory&oldid=461609995) (visited on 11/23/2011).
- [Wikd] *Inter-process communication*. Wikipedia. Nov. 2011. URL: [http://en.wikipedia.org/w/index.php?title=Inter-process\\_communication&oldid=459566980](http://en.wikipedia.org/w/index.php?title=Inter-process_communication&oldid=459566980) (visited on 11/23/2011).
- [Wike] *Internet socket*. Wikipedia. Nov. 2011. URL: [http://en.wikipedia.org/w/index.php?title=Internet\\_socket&oldid=460411478](http://en.wikipedia.org/w/index.php?title=Internet_socket&oldid=460411478) (visited on 11/23/2011).
- [Wikf] *Operating System*. Wikipedia. Nov. 2011. URL: [http://en.wikipedia.org/w/index.php?title=Operating\\_system&oldid=462073775](http://en.wikipedia.org/w/index.php?title=Operating_system&oldid=462073775) (visited on 11/23/2011).
- [Wikg] *PC*. Wikipedia. Nov. 2011. URL: [http://en.wikipedia.org/w/index.php?title=Personal\\_computer&oldid=461305927](http://en.wikipedia.org/w/index.php?title=Personal_computer&oldid=461305927) (visited on 11/23/2011).
- [Wikh] *Processor register*. Wikipedia. Nov. 2011. URL: [http://en.wikipedia.org/w/index.php?title=Processor\\_register&oldid=455430304](http://en.wikipedia.org/w/index.php?title=Processor_register&oldid=455430304) (visited on 11/23/2011).
- [Wiki] *Reduced instruction set computing*. Wikipedia. Nov. 2011. URL: [http://en.wikipedia.org/w/index.php?title=Reduced\\_instruction\\_set\\_computing&oldid=461872168](http://en.wikipedia.org/w/index.php?title=Reduced_instruction_set_computing&oldid=461872168) (visited on 11/23/2011).
- [Wikj] *Symbian*. Wikipedia. Nov. 2011. URL: <http://en.wikipedia.org/w/index.php?title=Symbian&oldid=461931600> (visited on 11/23/2011).



## *Bibliography*

---

- [Wikk] *Virtual machine*. Wikipedia. Nov. 2011. URL: [http://en.wikipedia.org/w/index.php?title=Virtual\\_machine&oldid=460282306](http://en.wikipedia.org/w/index.php?title=Virtual_machine&oldid=460282306) (visited on 11/23/2011).
- [Www] *Apple*. Website. July 2009. URL: <http://www.theapplemuseum.com/> (visited on 11/23/2011).

# Glossary

**AIK** (Attestation Identity Key) is an alias for the EK [Tpm, p.68].

**Apple** is a company providing operating systems, mobile devices and home and business computer [Www].

**ARM** is an abbreviation for a Arcon RISC Machine CPU.

**black-list** is a list of entries indicating not accepted entries.

**boot** describes the initial operations a computer performs when switched on [Wika].

**bootloader** manages the boot process [Wika].

**bytecode** is a collection of instructions for a Virtual Machine [Wikk].

**compile time** is the moment when source code of a program is being compiled into a binary file.

**copy-on-write** This mechanism is used to prevent unnecessary copy operations of one process to the other. Only if the child process tries to write on the address space shared by parent and child a copy operation is performed [Lov05, p.63].

**CPU** (Central Processing Unit) is the hardware which processes the instructions of a program [EK11a].

**daemon** is a service running in the background of the operating system.

**doubly linked list** Is a chained data structure where every element is connected with its predecessor and its successor.

**EK** (Endorsement Key) is a key used by the MTM to sign RIM-certificates 3.2.2 [Mtm, p.8, 29, 82], [Tpm, p.41, 67].

**exception** is an information about a process state. An exception is “thrown” to other application layers [Lov05, p.116].

**ext3** (third extended filesystem) a filesystem architecture that monitors changes done to files, which ensures consistency if a writing operation was interrupted by e.g. a systemcrash.

**FAT32** (File Allocation Table) a filesystem architecture.

**firmware** is a stored software inside the permanent memory of a hardware component [Wikb].

**flash** or flashing is the process where data or software is transmitted and stored into the permanent memory of a hardware component [Wikc].

**flashed image** is a software component stored into a permanent memory [Wikc].

**google** is an multinational internet service provider and a software developpment company [Goo11b].

**GPL** (GNU General Public License) forces programs/libraries using code written under GPL also to be released under this license [FSF11].

**hash** is a function that calculates a key value from a set of data. This key value is typically smaller in size than the size of the data. It can be used to accelerate sorting and searching operations [KKP09, Ch.5], [MOV01, ch.9.2.1].

**HMAC** (Hash based Message Authentication Code) [MOV01, ch.9.67].

**hook** describes a point where a specific function is called.

**integrity** is a metric that indicates a well known state asumed as healthy.

**inter process communication** is method utilized by processes and threads to communicate with each other. It is used to pass information, requests and results [Wikd].

**interrupt** is a state where the interrupting event perform its task imideatly preempting other actual running tasks [Lov05, p.115].

**IRQ** an interrupt request initiates an interrupt which cause the actual running task in the CPU to suspend its work and to let the interrupting task perform its work.

**kernel** is the core software component of the Linux operating system [Lov05, p.32].

**kill** is a signal sent to a process causing the process receiving it to stop its task immediately and to release its gained resources.

**Laptop** is a mobile PC with independent power supply.

**library** is a loadable commulation of additional functions loaded at runtime by a program.

**linux** is a free open source operating system [Lov05, p.31].

**lock** is used by the kernel to secure critical sections. Only one process can have the lock and enter the critical section. Other processes are forced to wait to enter the critical section until the process possessing the lock leaves this section and releases the lock.

**marshall** is the conversion of data structures into a format used to transfer these structures between communicating processes.

**memory-management-unit** arbitrates a virtual memory to the user. It protects the access to each virtual memory [Lov05, p.50].

**microsoft** is an operating system and software development company.

**MLTM** Mobile Local-owner Trusted Module is a MTM with local ownership setup.

**MRTM** Mobile Remote-owner Trusted Module is a MTM with remote ownership setup.

**MTM** (Mobile Trusted Module) is a hardware-based integrity assurance for mobile devices [Mtm].

**operating system** “An operating system is a set of programs that manage computer hardware resources and provide common services for application software” [Wikf].

**PC** (Personal Computer) is a small computer for individual personal use [Wikg].

**PCR** Platform Configuration Registers are immutable storing registers only accessible by the MTM.

**proxy** is a mediator service between different entities..

**RAM** (Random Access Memory) is a volatile memory which loses its stored information when powered off [EK11b].

**register** is a memory storage (hardware) inside the CPU [Wikh].

**remote attestation** Remote attestation describes a process where a service provider can attest an entity calling a service as not manipulated without physically access this entity.

**RIM** (Reference Integrity Metric) is a calculated metric of an entity with further information about this entity [Mtm, p. 18].

**RISC** (Reduced Instruction Set Computer) describes a CPUs design using a set of simple instructions [Wiki].

**RTM** Root-of-Trust-Measurement is a measurement module inside the MTM.

**RTV** (Root-of-Trust-for-Verification) is a verification module inside the MTM.

**runtime** is the time after the boot process until the power off. During this period of time non operating system software can be used.

**RVAI** (Root Verification Authority Identifier) is a key used by the MTM to verify new passed verification keys to it.

**scheduling** is a arbitration logic which manages the temporal execution of programs in operating systems [Lov05, p.73].

**SDCard** (Secure Digital Memory Card) a memory chip stored on a removeable card to flash data to it and to read the data from the card.

**SHA256** (Secure Hash Value) is a cryptographic hash representing a unique checksum of 256 bit length [MOV01, ch.9.52].

**shared secret** is a secret known to all participating communication endpoints , such as a password or a secret key.

**sign** is a mathematical operation which assures the origin of a message/data [MOV01, ch.11.1].

**signal** is a short defined operating system message for processes.

**socket** is a data communication endpoint for exchanging data between processes [Wike].

**SRK** (Storage Root Key) is a key used by the MTM to sign keys.

**stack** is a memory where the last set (written) data is the first data received from it when read.

**starvation** is the situation where a task is always preempted from execution by other tasks.

**stub** is a piece of code acting as a substitute for full implementation of a function or a program.

**swap space** is a memory available on a non volatile storage. It is used to temporary store data from RAM to gain free memory for other purposes/tasks.

**Symbian** is an operating system for mobile devices [Wikj].

**system-call** is a set of functions arbitrated by the kernel to the user [Lov05, p.102].

**TAPL** (Trusted Application List) contains all calculated and signed hash values of trusted applications.

**TCG** Trusted Computing Group is an organisation for standards.

**TPM** Trusted Platform Module is an integrity measurement and verification module for desktop PCs and Laptops [Tpm].

**trust anchor** describes a reliable well known and trusted state from which a latter state can extend.

**tuple** is a list of congeneric entires.

**USB** (Universal Serial Bus) is a connection standard for various external devices [UIF11].

**verified** is a mathematical operation which verifies the origin of a message/data [MOV01, ch.11.1].

**verifies** is a mathematical operation which verifies the origin of a message/data [MOV01, ch.11.1].

**verify** is a mathematical operation which verifies the origin of a message/data [MOV01, ch.11.1].

**Virtual Machine** is a virtual computer with software representations of hardware components [Wikk].

**virtual memory** is a possibly phisicly fragmnted memory pretending to processes being coherent and starting with the adress 0.

**white-list** is a list of entries indicating accepted entries.

**zygote** is a core process in the Android OS which forks new Dalvik Virtual Machines to start new Android processes (Activities).

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

Hamburg, 14. December 2011

---

Martin Landsmann