Fachhochschule Hannover
University of Applied Sciences and Arts
Fakultät IV – Wirtschaft und Informatik

Integration von Trusted Computing Technologien in die Android-Plattform

Masterarbeit im Studiengang Angewandte Informatik in der Abteilung Informatik der Fakultät IV an der Fachhochschule Hannover

Johannes Westhuis
Matrikelnummer 1092613

Donnerstag, 19. August 2010

Selbstständigkeitserklärung

Hiermit bestätige ich, dass ich die eingereichte Masterarbeit selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum

Unterschrift

Autor
Herr
Johannes Westhuis B. Sc.

Fachhochschule Hannover
Fakultät IV
Abteilung Informatik
Ricklinger Stadtweg 120

30459 Hannover

Erstprüfer
Herr Prof. Dr. rer. nat.
Josef von Helden

Fachhochschule Hannover
Fakultät IV
Abteilung Informatik
Ricklinger Stadtweg 120

30459 Hannover

Zweitprüfer
Herr Dr.
Carsten Rudolph

Fraunhofer Institute
for Secure
Information Technology
Rheinstrasse 75

64295 Darmstadt

Danksagung

Ich danke meinen Prüfern, Prof. Dr. Josef von Helden und Dr. Carsten Rudolph für die Bewertung dieser Arbeit. Weiterhin danke ich meinen Betreuern Ingo Bente, Nicolai Kuntze und Jörg Vieweg für die Zeit und den großen Einsatz. Besonders danken möchte ich meiner Freundin, meiner Familie und meinen Freunden, die während dieser Zeit viel Unterstützung geboten haben. Desweiteren danke ich noch den vielen Korrekturlesern.

Inhaltsverzeichnis

1. Einleitung und Motivation

Diese Masterarbeit befasst sich mit der Integration von Trusted Computing Technologien in die Android Plattform. In dem Abschnitt Einleitung wird beschrieben, warum eine solche Integration benötigt wird. Anschließend werden Arbeiten, die sich mit diesem oder ähnlichen Themen befassen, aufgezeigt.

1.1. Einleitung

Die Struktur von Firmennetzwerken ist vielen Veränderungen unterworfen. Durch den technischen Fortschritt und die Einführung neuer Konzepte entwickeln sie sich ständig weiter. Daraus ergab sich beispielsweise eine Entwicklung der Rechnerlandschaften von Firmen. Diese führte von Mainframe-Servern über Client-Server-Architekturen mit relativ homogenen Geräten bis hin zu Netzwerken von verschiedensten, auch mobilen Komponenten.

Dabei verändert sich nicht nur die Architektur des Netzwerkes. Weiterhin ändern sich die Sicherheitsfragen, die sich für ein solches Netzwerk ergeben. Frühe Mainframe-Server wurden von dafür verantwortlichen Personen bedient, welche die Prozesse von Benutzern ausführten. Die Bedrohungen gingen dabei vor allem von manipulierten oder fehlerhaften Programmen der Benutzer aus. Die bestimmenden Sicherheitsüberlegungen betrafen daher vor allem den Schutz der Benutzerprozesse und Ressourcen vor einem Zugriff von anderen Benutzern sowie die Authentisierung und Autorisierung einzelner Benutzer und Gruppen.

Diese Probleme sind auch bei allen späteren Netzwerkumgebungen anzutreffen, allerdings müssen noch weitere Probleme gelöst werden. Client-Server-Umgebungen erfordern sichere Netzwerkprotokolle, die die Integrität und Authentizität der übertragenen Nachrichten sicherstellen. In diesen Umgebungen ist es möglich, dass Clients den Netzwerkverkehr abhören oder manipulieren.

In heutigen Umgebungen bestehen die Netzwerkteilnehmer nicht mehr nur aus durch die Netzwerkadministratoren kontrollierten Geräten, sondern aus einer Vielzahl mobiler Geräte. Diese reichen von Notebooks der Außendienstmitarbeiter bis hin zu Handheld-Geräten und modernen Mobiltelefonen, die über einen Zugang zu dem Netzwerk verfügen.

Neben die Frage nach der Vertrauenswürdigkeit des Benutzers tritt nun immer mehr auch die Frage nach der Vertrauenswürdigkeit des Endgerätes. Ein manipuliertes Endgerät ermöglicht einem Angreifer Zugang zu vertraulichen Daten seines Ziels. Dies könnte auch durch eine Einschleusung oder Bestechung eines Mitarbeiters der Zielfirma erreicht werden. Durch den direkten Kontakt mit den Mitarbeitern einer Firma ist dabei allerdings das Risiko, entdeckt und belangt zu werden, höher als bei einem manipulierten

Gerät, welches unauffällig Daten an unbekannte Orte versendet.

Mobile Geräte stellen ein leichtes Ziel dar, da sie häufig unbeaufsichtigt gelassen werden. Weiterhin lässt sich die Soft- und Hardware eines solchen Systems durch den Benutzer nicht überprüfen, was es schwierig macht, effektive Schutzmaßnahmen zu treffen.

Im Gegensatz zu klassischen, mobilen Plattformen, bei denen die Hard- und Software von einem bestimmten Hersteller kam und Erweiterungen an diesen Geräten nur in minimalem Umfang möglich waren, entwickeln sich moderne Geräte zu offenen Systemen, die Anwendungen von beliebigen Herstellern zulassen. Jeder Benutzer eines Systems ist in der Lage, auch von nicht vertrauenswürdigen Quellen Programme zu installieren. Die dadurch entstehende Gefahr einer Kompromittierung der Plattform durch Malware ist im Vergleich zu den bisherigen, von Herstellern kontrollierten Geräten deutlich angestiegen.

Die oben genannten Entwicklungen führen zu folgenden Problemen:

- hohes Verlustrisiko der mobilen Geräte
- einfache Einschleusung manipulierter Software durch erweiterbare Geräte
- keine Möglichkeit für den Benutzer, Manipulationen zu erkennen

Die Vielseitigkeit und Geschwindigkeit dieser neuen Geräte birgt aber nicht nur Gefahren. Denn durch diese Eigenschaften bieten sich auch Möglichkeiten, den Bedrohungen entgegenzuwirken.

Diese Arbeit untersucht, wie die Ansätze des Trusted Computing auf mobile Geräte angewandt werden können. Dabei wird anhand des Open-Source-Betriebssystems Android ein Konzept erstellt, mit dem diese Technologien integriert werden können.

1.2. Related Work

Bei Trusted Computing handelt es sich um einen Ansatz, Aussagen über den Zustand verschiedener Geräte zu erlangen. Diese Aussagen lassen sich dann nutzen, um die Vertrauenswürdigkeit des Geräts einschätzen zu können. Die Trusted Computing Group (TCG)[1] definiert die Spezifikationen auf denen Trusted Computing basiert [24]. Es gibt aber neben den Beiträgen der TCG noch weitere Arbeiten in diesem Bereich.

Löhr et. al definieren Patterns, mit denen grundlegende Trusted-Computing-Techniken beschrieben werden [13]. Eine weitere Ausarbeitung befasst sich mit der Verschlüsselung von Daten mit Hilfe von Trusted Computing [29]. Diese Technologien können verwendet werden um die in Abschnitt 5 beschriebenen Konzepte zu sichern und zu ergänzen.

Dietrich und Winter beschreiben eine Architektur, mit welcher der sichere Bootvorgang des Trusted Computings auf mobilen Plattformen unterstützt werden kann [9]. Ihr Ansatz diskutiert den Einsatz von Software-basierten Mobile Trusted Modules (MTM), die von der TCG spezifiziert wurden. Dieser Ansatz kann in die hier präsentierte Lösung integriert werden, um sie zu verbessern (Siehe 5.7).

[1] www.trustedcomputinggroup.org

Shabtai, Fledel und Elovici [21] zeigen den Einsatz von SELinux auf Android Geräten. SELinux erweitert die Standard-Sicherheitsmechanismen der Android Plattform durch die Nutzung von feingranularen Zugriffsmodellen. Dieser sehr vielversprechende Ansatz kann parallel mit der hier vorgestellten Lösung verwendet werden und erhöht durch die definierten Zugriffsmodelle die allgemeine Systemsicherheit.

Nauman et. al präsentieren ihre eigene Architektur, mit der, ähnlich wie bei der hier vorgestellten Lösung, eine Trusted-Computing-Technologie auf der Android Plattform etabliert werden kann [16]. Diese Lösung wird im Abschnitt 5 als Alternative näher betrachtet und mit der eigenen Entwicklung verglichen.

Insgesamt sind viele Quellen zu den Grundlagen des Trusted Computing und der Anwendung dieser Technologien auf verschiedene Plattformen zu finden. Zu der Integration in die Android-Plattform, konnte bisher nur der Aufsatz von Nauman gefunden werdern. Der Themenkomplex erfordert aufgrund der beschriebenen Aktualität eine genauere Betrachtung, welche durch diese Arbeit gegeben werden soll.

2. Trusted Computing

Chris J. Mitchell [15] beschreibt, welche Motivation hinter den Bemühungen steht, Trusted Computing für physische Geräte zu etablieren:

> The traditional assumptions regarding the physical security of important computer systems are clearly completely inappropriate for the vast majority of PCs in use today. Most such PCs have only a single user, and no physical security is provided for the PC hardware. Short-term access to the PC can easily result in unauthorised copying of information and/or modifications to software and data[...]. That is, regardless of how 'secure' the operating system is in the traditional sense, the lack of guarantees about physical security means that the correctness of software or information on the PC cannot be trusted; neither can the confidentiality of any such information.

Mobile Geräte sind den beschriebenen Gefahren deutlich häufiger ausgesetzt, da sie sich nicht an einem festen Ort befinden, der gegen unbefugten Zugang gesichert werden kann. Sie werden transportiert und bewegen sich meist in Gebieten, in denen keine Sicherungsmöglichkeit besteht. Ein gezielter Taschendiebstahl oder das Ausnutzen einer Unachtsamkeit des Benutzers des Gerätes ermöglichen es Angreifern recht einfach, in den Besitz des mobilen Gerätes zu gelangen. Es wird eine physikalische Komponente benötigt, um Aussagen zur Sicherheit des Gerätes treffen zu können. Jede Softwarelösung allein kann einer Manipulation des Gerätes nicht standhalten. Eine Hardwaresicherung hingegen lässt sich nur schwer manipulieren ohne Spuren zu hinterlassen.

Die Trusted Computing Group spezifiziert neben einer Hardwarekomponente verschiedene Protokolle und Mechanismen, welche insgesamt die Sicherheit von Plattformen verbessern sollen. Sie definiert außerdem die Bedeutung des Begriffs *trust* in Computersystemen.

In diesem Kapitel wird erläutert was die Definition von *trust* ist und wie mit ihr der Umgang mit einer Plattform verbessern werden kann. Weiterhin wird die Hardwarekomponente, das Trusted Platform Module, erklärt. Anschließend werden weitere grundlegende Konzepte der TCG beschrieben, welche in dieser Masterarbeit genutzt werden.

2.1. Trust

Die Trusted Computing Group definiert *trust* als [10] [11]:

> An entity can be trusted if it always behaves in the expected manner for the intended purpose

Wenn also ein Programm sich so verhält wie man es erwartet, kann ihm vertraut werden. Eine der Fragen die sich daraus ergeben ist: „Wann verhält sich das Programm, wie ich es erwarte?“ Diese Frage kann nur gelöst werden, wenn eindeutig bestimmt werden kann, um welches Programm es sich handelt. Wenn das Programm *identifiziert* werden kann, kann diesem Programm ein bestimmtes Verhalten zugesichert werden.

Dabei ist unwichtig, ob dieses Verhalten ein positives oder negatives Ergebnis für das System nach sich zieht. Dies wird auch von David Grawrock [11] beschrieben:

> Relying entities do trust the infected platform to perform bad operations. As a result, trust does not always equals goodness. Just because the relying party is going to trust the platform does not mean that the relying party trusts the platform to do a good job. The opposite is true; the relying party may trust the platform to do a bad job.

Wenn allerdings bestimmt werden kann, ob es sich bei einem Programm um einen Virus handelt oder ob es durch einen Angreifer manipuliert wurde, wird anhand der Identifizierung eine Entscheidung über den Sicherheitszustand möglich.

2.2. Measurement

Wie beschrieben muss eine Identifizierung der Komponenten durchgeführt werden. Dies kann durch eine Erstellung eines Hashwertes für die jeweilige Komponente erreicht werden. Ein solcher Hashwert benötigt mindestens eine schwache Kollisionsresistenz [20]. Es soll zu einem gegebenen Objekt O mit dem Hashwert H kein Objekt O_1 gefunden werden können, welches einen Hash H_1 besitzt, bei dem $H = H_1$ gilt.

Wenn dies nicht gegeben ist, wäre es für einen Angreifer möglich, zu einem berechneten Hashwert einer Anwendung ein Programm zu entwickeln, welches den gleichen Hashwert der Binärdatei besitzt, jedoch eine andere Funktion ausführt. Diese beiden Programme wären dann nicht zu unterscheiden.

In der aktuellen TCG Spezifikation ist für die Erstellung eines Hashwertes der SHA-1 Algorithmus vorgesehen. Da dieser Algorithmus allerdings als gebrochen gilt [27] und daher in Zukunft nicht mehr sicher sein wird, wird in kommenden Spezifikationen eine Erweiterung der möglichen Algorithmen nötig sein. Die Erstellung eines Hashwertes wird auch als Measurement oder Messung bezeichnet.

2.3. Trusted Platform Module

Bei dem Trusted Platform Module (TPM) handelt es sich um einen Hardware-Chip, welcher in der Lage ist, verschiedene kryptographische Operationen durchzuführen. Weiterhin besitzt es geschützte Speicherbereiche, in denen zum Beispiel Schlüsselmaterialien abgelegt werden können. Das in der Spezifikation[24] beschriebene grundlegende Design des TPMs wird in Abbildung 2.1 dargestellt.

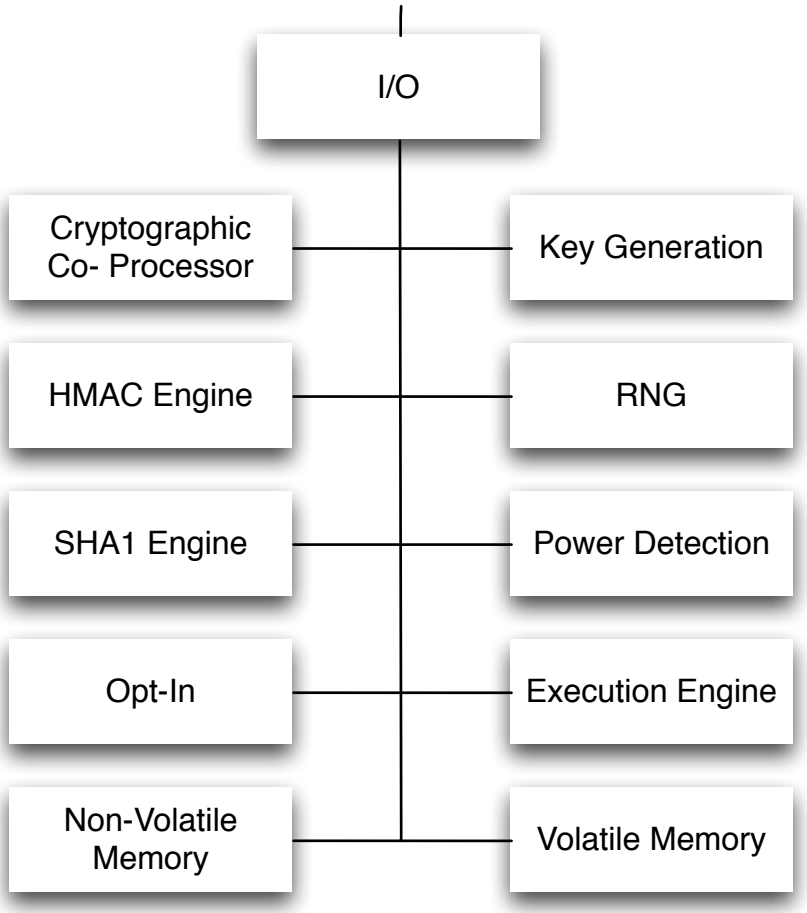

Abbildung 2.1.: TPM Architektur der TCG[24]

Die Abbildung zeigt, dass das TPM aus verschiedenen Komponenten besteht, welche jeweils eine besondere Funktion erfüllen. Die I/O-Komponente wird genutzt um die Kommunikation mit dem TPM zu steuern. In konventionellen PC-Systemen wird die I/O Komponente beispielsweise über den LPC-Bus angesprochen. Für diesen Bus spricht, dass er schon früh während des Systemstarts verfügbar ist. Weiterhin werden kryptographische Operationen bereitgestellt. Die RNG Komponente ist für die Erzeugung von echten Zufallszahlen zuständig. Die SHA1-Engine erstellt vor allem Hash-Werte für die Operationen innerhalb des TPMs. Sie kann aber auch von außenstehenden Komponenten für die Erstellung von Hashwerten verwendet werden [24]. Weiterhin erlauben es

die Bausteine Ver- und Entschlüsselungen durchzuführen sowie Schlüssel zu verwalten. Die Power-Detection Komponente meldet Änderungen an der Stromversorgung und mit der Opt-In Komponente lässt sich das TPM aktivieren/deaktivieren und verschiedene Zustände verwalten[24].

Root of Trust

Jedes TPM besitzt ein eindeutiges Schlüsselpaar, den so genannten Endorsement Key (EK). Der private Teil dieses RSA-Schlüssels verlässt das TPM nie und wird nur in besonderen Fällen ausgetauscht. Mit diesem Schlüssel ist eine Identifizierung des TPMs möglich.

Ein weiteres wichtiges Schlüsselpaar des TPM ist der Storage Root Key (SRK). Dieser ermöglicht das Verschlüsseln und sichere Ablegen von Daten auf der Festplatte. Mit Hilfe des SRKs können auch weitere Schlüssel erzeugt und gelagert werden.

Diese beiden Schlüsselpaare sind wichtig für eines der Hauptkonzepte des Trusted Computing. Sie bilden eine Root of Trust. Eine Root of Trust stellt einen im System vorhandenen Vertrauensanker dar.

Der Endorsement Key ist zum Beispiel für Außenstehende der Vertrauensanker, da er beweisen kann, das es sich um ein System mit einem TPM handelt. Er ist eindeutig einem TPM zugeordnet und kann nur zum Entschlüsseln von Nachrichten verwendet werden. Wenn eine Nachricht mit dem öffentlichen Teil des EKs verschlüsselt wurde, kann diese nur durch das TPM gelesen werden. Es werden weitere Schlüssel generiert, über welche der Zustand des Systems attestiert werden kann. Diese Schlüssel werden über den EK an die Plattform gebunden. Da über den Endorsement Key sichergestellt wird, um welches TPM es sich handelt, wird dieser auch Root of Trust for Reporting (RTR) genannt.

Der Storage Root Key verlässt die Platform nie und wird nur gewechselt, wenn der Besitzer des TPMs geändert wird. Damit stellt er sicher, dass Daten, welche mit ihm verschlüsselt wurden nur auf dem System, auf dem sie verschlüsselt wurden, entschlüsselt werden können. Aus diesem Grund wird der SRK als Root of Trust for Storage (RTS) bezeichnet.

Platform Configuration Register

Das TPM besitzt einige Platform Configuration Register (PCR). Diese Register haben den Zweck, die Messungen der Plattform abzulegen. Da die Messungen für Entscheidungen zur Vertrauenswürdigkeit eines Systems von höchster Wichtigkeit sind, ist es auch besonders wichtig, dass ein PCR nicht manipuliert werden kann und dass Angreifer nicht in der Lage sein dürfen, ein PCR zu fälschen. Weiterhin ist der Speicherplatz eines TPMs stark begrenzt. Die TCG bietet für diese beiden Probleme eine geschickte Lösung an: Durch die Hardware des TPMs wird nur eine Funktion bereitgestellt, um in ein PCR zu schreiben. Mit der *extend* Operation wird das PCR in einer genau definierten Weise beschrieben.

In der „TPM Main Specification Part 1: Design Principles“ beschreibt die TCG [24] die Funktion der *extend* Operation wie folgt:

> The PCR is designed to hold an unlimited number of measurements in the register. It does this by using a cryptographic hash and hashing all updates to a PCR. The pseudo code for this is:
> PCRi New = HASH (PCRi Old value || value to add)

Der Wert eines PCRs besteht also immer aus dem Hashwert des vorherigen Wertes, welcher mit dem hinzuzufügenden Wert verknüpft wurde. Daraus ergibt sich die Eigenschaft, dass der PCR immer die gleiche Länge benötigt, da das Ergebnis einer Hash Funktion immer eine feste Länge besitzt. Dies reduziert für eine beliebige Anzahl von Messungen den Speicherbedarf immer auf diese feste Länge.

Ein weiterer Vorteil für die Sicherheit ergibt sich daraus, dass es für den Angreifer nicht möglich ist, einen bestimmten Wert in das PCR zu schreiben. Er kann immer nur etwas hinzufügen. Und da durch die Hashfunktion sichergestellt ist, dass eine Kollision nicht in endlicher Zeit erzeugt werden kann, ist es ihm nicht möglich einen von ihm gewünschten Wert im PCR zu erzeugen.

Durch diese Funktion ergibt sich eine Abhängigkeit von der Reihenfolge der Messungen. Das Ergebnis des Hashwertes von zwei Dateien ist zwar immer jeweils gleich, jedoch führt die Verknüpfung mit dem Vorgänger zu dieser Abhängigkeit. Dieses Verhalten wird in Listing 2.1 dargestellt:

Listing 2.1: Abhängigkeit von der Reihenfolge

```
# tpm_readpcrs 11
11:0000000000000000000000000000000000000000
# tpm_readpcrs 12
12:0000000000000000000000000000000000000000
// Werte in PCR 11 einfuegen
# tpm_extendpcrs 11 12345678901234567890
# tpm_extendpcrs 11 09876543210987654321
// Werte in umgekehrter Reihenfolge in PCR 12 einfuegen
# tpm_extendpcrs 12 09876543210987654321
# tpm_extendpcrs 12 12345678901234567890
// Ausgabe der Ergebnisse
# tpm_readpcrs 11
11:E5D5490F0E23A71D024ADC9E4D024BF6DB97C627
# tpm_readpcrs 12
12:3811DE42064671B7F20C30DD742866EDA44CD4B9
```

In diesem Listing wurden in zwei gleich vorbelegte PCRs die gleichen Werte in unterschiedlicher Reihenfolge eingefügt. Als Ergebnis lassen sich zwei unterschiedliche Werte in den PCRs feststellen.

Um die gespeicherten Messungen, die wie beschrieben durch die *extend* Operation zu einen Wert zusammengefasst sind, überprüfen zu können, sind Informationen über die durchgeführten Messungen und deren Reihenfolge notwendig.

Stored Measurement Log

Für diesen Zweck wird außerhalb des TPMs das Stored Measurement Log (SML) gespeichert.

Das SML enthält eine Liste der gemessenen Dateien und deren Messwert. Diese Liste kann genutzt werden, um das PCR zu errechnen. Ergibt sich nach der Berechnung der Werte nicht der PCR-Wert, dann hat eine Manipulation stattgefunden. Es kann nicht genau festgestellt werden, welche Dateien manipuliert wurden, dies ist jedoch für eine Vertrauensentscheidung nicht nötig. Wenn eine Manipulation stattgefunden hat, fällt diese Entscheidung negativ aus.

Daher kann das SML auch außerhalb der vom TPM geschützten Umgebung gespeichert werden. Wird das SML verändert, wird die Berechnung der Werte nicht zu dem PCR-Wert passen, da es in endlicher Zeit nicht möglich ist, das PCR mit einem entsprechenden Wert zu belegen.

2.4. Attestation

Die durch Messungen erzeugten Werte bieten allein noch keinen Vorteil für den Benutzer einer Plattform. Es ist ihm unmöglich, aus einem 20-Byte-Messwert eine Entscheidung über den Zustand eines Programmes zu treffen. Daher gibt es einen Mechanismus, der Attestation genannt wird. Bei diesem werden Messwerte und andere Informationen an einen Verifier gesendet. Dieser Verifier überprüft die Werte anhand einer Policy und trifft daraufhin eine Entscheidung.

Da dem Verifier Informationen über das gesamte System gesendet werden und diese über die RTR dem TPM zugeordnet werden können, gab es einige Bedenken, welche den Datenschutz des Benutzers betrafen. Über diese Zuordnung könnte bei jeder Verifizierung ein genaues Bild über alle Aktivitäten eines Rechners gezeichnet werden.

Dies stellt eine erhebliche Einschränkung der Privatsphäre des Benutzers dar. Aus diesem Grund wurden zwei Verfahren entwickelt, welche die genaue Zuordnung eines TPMs zu einer Verifikation verhindern sollen: Attestation Identity Keys und Direct Anonymous Attestation.

Attestation Identity Keys (AIK) können im TPM mit dem Endorsement Key erzeugt werden. Sie werden in der TCG TPM Main Specification [24] definiert um die Signierung der Verifikationsnachrichten durchzuführen. Da ein TPM beliebig viele AIKs erstellen und nutzen kann, ist so eine Korrelation der Schlüssel mit den Aktivitäten nicht mehr möglich. Da nun allerdings sichergestellt werden muss, dass der AIK zu einem TPM gehört, gibt es sogenannte Certificate Authorities (CA). Diese signieren das AIK. Die CA schickt das signierte AIK wieder an den Rechner zurück, verschlüsselt mit dem Endorsement Key des Anfragenden. So kann sichergestellt werden, dass ein AIK nur von der anfragenden Plattform benutzt wird.

Es entsteht allerdings ein Problem mit der Privatsphäre, wenn der Verifier und die CA zusammenarbeiten. Die CA kann eine Korrelation zwischen den signierten AIKs und den mitgesendeten Endorsement Keys erzeugen, da ihr diese beiden Attribute bekannt sind.

Dieses Verhalten kann gewollt sein, um beispielsweise einen bestimmten AIK ungültig zu machen, sobald bekannt wird, dass ein TPM kompromittiert worden ist. Ein TPM könnte durch Hardwareangriffe den privaten Teil des EKs preisgeben. In einem solchen Fall sollte es möglich sein, die dazugehörigen Schlüssel zu deaktivieren. Allerdings ist es so möglich, über die Zuordnung von genutzten AIKs zu EKs und mit der Hilfe der CA die Aktionen eines Benutzers nachzuvollziehen.

Es existiert ein weiteres Verfahren, welches versucht diese Problematik zu lösen. Direct Anonymous Attestation (DAA) wird von Brickell, Camenisch und Chen [8] beschrieben. Andreas Pashalidis [18] fasst das Ziel der DAA wie folgt zusammen:

> This mechanism, called 'Direct Anonymous Attestation' (DAA), does not require an authority to certify each AIK. Instead, the TPM is required to obtain a digital 'attestation certificate' from an Issuer (typically the TPM or TP manufacturer) only once. This certificate can then be used by the TPM to digitally sign an arbitrary message in such a way that (a) the verifier is convinced that the message was generated by a TCG conformant TPM, and (b) the resulting signature cannot be linked to any other DAA signature that was previously generated by this particular TPM

Auf dem Gerät wird ein Geheimnis erzeugt, welches dann von einer vertrauenswürdigen Instanz signiert wird. Das Besondere an diesem Verfahren ist, dass das Geheimnis das TPM nie verlässt, die signierende Instanz es daher auch nicht kennen darf. Um dies zu ermöglichen, beweist das TPM mit einem „proof of knowledge“ [18], dass es das Geheimnis kennt. Anschließend wird dann eine „blinde“ Signatur durch die vertrauenswürdige Instanz durchgeführt, bei der wiederum das Geheimnis das TPM nicht im Klartext verlässt.

Dem Verifier kann das TPM anschließend beweisen, dass es das Geheimnis kennt, weiterhin ohne einen Teil des Geheimnisses preiszugeben. So kann mit diesem komplexen Verfahren für die Attestation sichergestellt werden, dass es sich bei der die Nachricht sendenden Einheit um ein echtes TPM handelt, mit dem Vorteil, dass keine Korrelationen zwischen den Aktionen des Benutzers hergestellt werden können.

Ein weiterer großer Vorteil dieses Verfahrens besteht darin, dass die Plattform nur einmal eine Verbindung mit einer vertrauenswürdigen Instanz aufbauen muss, um anschließend beliebige Nachrichten verschicken zu können, ohne dass die Interaktionen einer Plattform zurückverfolgbar sind. Beide Verfahren werden durch das TPM angeboten und können eingesetzt werden.

Trusted Network Connect

Ein Verfahren, welches Remote Attestation einsetzt um Netzwerkzugriff zu ermöglichen, ist Trusted Network Connect (TNC). TNC ist ein von der Trusted Computing Group spezifiziertes Protokoll, das verschiedene Komponenten auf mehreren Schichten definiert. Abbildung 2.2 stellt die in der TNC Spezifikation [26] definierte Architektur dar.

Die oberste Schicht setzt sich aus zwei Komponenten zusammen, Integrity Measurement Collectors (IMC) und Integrity Measurement Verifiers (IMV). Je ein IMC und ein

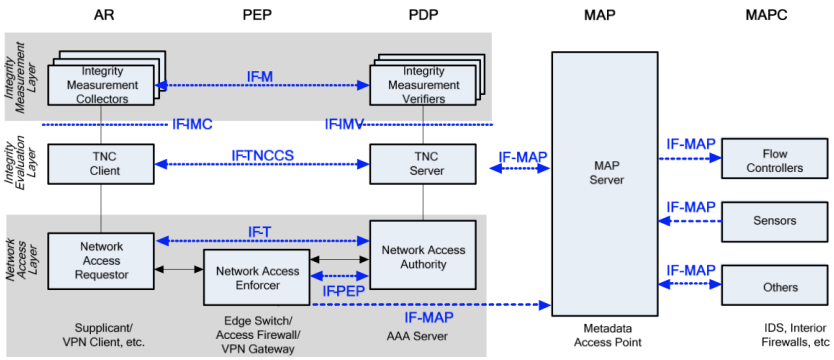

Abbildung 2.2.: TNC Architektur der TCG [26]

IMV bilden ein Paar. Der IMC wird auf dem Client genutzt, um Messungen beliebiger Art durchzuführen. Der entsprechende IMV überprüft die Messungen dann anhand einer Policy. Ein mögliches Beispiel für ein IMC/IMV Paar ist ein IMC, welcher alle offenen Ports einer Plattform überprüft und an den dazu gehörenden IMV sendet, welcher entscheiden kann welche Ports erlaubt sind und welche nicht. Sind auf dem Client Ports geöffnet, die gegen die Policy verstoßen, wird kein Zugang zum Netzwerk ermöglicht.

Die unteren Schichten sorgen für den Transport der Nachrichten und den Zugang zu dem Netzwerk. Dabei wird zum Beispiel der Netzwerkzugang über das 802.1x Protokoll etabliert. Die IEEE [12] definiert mit dem 802.1x-Protokoll, wie ein authentisierter und autorisierter Netzzugang ermöglicht werden kann.

2.5. Chain of Trust

Die für die Attestation notwendigen Informationen lassen sich wie beschrieben durch Messungen generieren. Es ist allerdings notwendig, diese Messungen vor Manipulationen zu schützen. Dazu ist es besonders wichtig, dass Komponenten sich nicht selber messen dürfen, da sie sonst in der Lage wären, falsche Tatsachen vorzuspielen. Eine kompromittierte Komponente würde bei einer solchen Messung statt des echten Messwerts einen gefälschten herausgeben, welcher mit dem einer gültigen Konfiguration übereinstimmt. Daher ist es nötig, die Messung immer durchzuführen, bevor die Kontrolle an die Komponente übergeht.

Zu diesem Zweck gibt es die Chain of Trust. In einer Chain of Trust wird angenommen, dass ein transitives Vertrauensverhältnis [23] besteht. Das bedeutet, dass wenn Person A einer Person B das Vertrauen ausspricht und Person A vertraut wird, dann kann auch Person B vertraut werden.

Da es immer eine Komponente geben muss, die am Anfang einer solchen Kette steht, muss dieser vertraut werden. Aus diesen Grund gibt es die Root of Trust for Measurement (RTM). Die RTM ist ein Codeblock, welchem vertraut werden muss und der die erste Messung durchführt. In konventionellen Geräten wird sie als Teil des BIOS ausgeliefert und misst dieses bei Systemstart.

Das BIOS führt anschließend eine Messung des Bootloaders des Betriebssystems durch und gibt die Kontrolle an diesen weiter. Anschließend wird durch den Bootloader ein Messwert des Betriebssystems erstellt, bevor dieses gestartet wird. Es gibt verschiedene Ansätze, um die Messungen des Systems vollständig zu halten. So werden alle gestarteten Anwendungen gemessen [19]. Alternativ misst ein Hypervisor verschiedene gestartete virtuelle Maschinen als Ganzes [4].

2.6. Mobile Trusted Module

Die Mobile Phone Working Group der TCG definiert eine Spezifikation [25] für Trusted Platform Module auf mobilen Plattformen. In dieser werden spezielle Mechanismen beschrieben, mit welchen das TPM erweitert wird. Allerdings existiert bisher keine, die Spezifikation realisierende Lösung, weshalb sie für die Betrachtung der Implementierung vorerst nicht in Frage kommt. Dennoch wird ein kurzer Einblick in die Erweiterungen gegenüber einem TPM aufgezeigt. Weiterhin werden die entwickelten Konzepte in Bezug auf diese Spezifikation betrachtet.

Die TCG Mobile Trusted Module Spezifikation enthält eine eigene Definition von Methoden für die lokale Verifizierung. Diese sieht vor, dass spezielle Komponenten des Systems die Verifikation der Messwerte vornehmen. Dazu werden verschiedene neue Komponenten spezifiziert, welche in den folgenden Abschnitten beschrieben werden.

2.6.1. Root of Trust for Verification

Die Mobile Trusted Module Spezifikation definiert einen weitere Root of Trust. Diese wird RTM+RTV genannt (Root of Trust for Measurement und Root of Trust for Verification). Sie ist, wie in Abschnitt 2.5 beschrieben, für die Grundlage der Chain of Trust notwendig. Zusätzlich wird bei jeder Messung auch eine Verifikation der gemessenen Werte durchgeführt.

Die Spezifikation [25] beschreibt, dass die RTM+RTV-Komponente in einem System zuerst gestartet werden soll. Nachdem sie einen Selbsttest durchgeführt hat, wird sie die Kontrolle an einen Measurement- und Verification-Agent übergeben, welcher nach einer Messung und Verifikation des Betriebssystems dann die Kontrolle an dieses übergibt.

Diese Verifikation dient vor allem zur einfachen Abbildung eines sicheren Boot-Mechanismus. Die beiden Vertrauensanker sollten als eine Einheit von dem Produzenten der mobilen Plattform bereitgestellt werden, um Manipulationen an den Messungen zu vermeiden.

Für den Verifizierer müssen allerdings gültige Werte des zu messenden Objekts bekannt sein, damit eine Verifikation durchgeführt werden kann. Diese müssen durch den Hersteller einer RTV bereitgestellt werden. Um gültige Messwerte abzuspeichern, sieht die Spezifikation die Ablage von Reference Integrity Metrics (RIM) vor.

2.6.2. Reference Integrity Metric

Die in der Spezifikation [25] definierte Möglichkeit, Messwerte vorzuhalten wird Reference Integrity Metric genannt und wie folgt beschrieben:

> A standard method is defined to provide Reference Integrity Metrics (RIMs) for use by the MTM. A RIM is a reference value to compare a measurement against. As an example, a RIM could be the SHA1 hash of a software image. A RIM Certificate ("RIM Cert") is an authenticated and integrity- protected structure containing a RIM and some auxiliary information.

Weiterhin bietet die TCG in dem Mobile Trusted Module einige Operationen und Strukturen an, mit denen diese RIM Certificates genutzt werden können. Eine eigene Schlüsselhierarchie von TPM Verification Keys kann von dem TPM aufgebaut werden, um die RIMs zu signieren. Die Schlüssel werden an Einheiten verteilt, welche RIM_Auth genannt werden. Ein so genannter RIM_Auth führt die Verifikation der Messwerte, beispielsweise innerhalb einer Chain of Trust, durch.

Dieser signiert dann das RIM und ruft die *MTM_VerifyRIMCertAndExtend*-Funktion des MTMs auf, um die gemessenen Werte in das MTM zu laden. Im Gegensatz zu den TPM-Funktionen, wird hier nur ein *extend* durchgeführt, wenn es mit einem gültigen Zertifikat signiert wurde. So soll sichergestellt werden, dass eine vertrauenswürdige Instanz wie die RTV die Messung verifiziert hat.

Das MTM muss also überprüfen, ob das RIMCert von einem vertrauenswürdigen RIM_Auth kommt, welcher einen gültigen TPM Verification Key (VK) besitzt. Um festzustellen ob ein VK gültig ist, muss dieser in dem MTM registriert werden. Dies kann auf verscheidene Arten geschehen: Ein VK, welcher vom MTM Besitzer registriert wird, ist ebenso gültig wie ein VK, welcher mit einem anderen, gültigen VK signiert wurde. Weiterhin gelten auch VKs, deren Hashwert im MTM gespeichert ist oder die vor dem Zeitpunkt von Integritätsmessungen in das MTM geladen wurden.

In der Spezifikation[25] wird beschrieben, dass das MTM selbst keine Verifikation durchführen muss:

> When running MTM_InstallRIM, there is no requirement for the MTM to itself verify the integrityData on the input rimCert (i.e. the integrity data accompanying the "external" RIM certificate). If the command parameters verify successfully, the MTM can assume that the relevant RIM has already been validated and authorized by the party that owns the verificationAuth data.

3. Analyse der Android Plattform

Das Betriebssystem eines Mobiltelefons muss verschiedenste Anforderungen erfüllen. Durch ihre geringe Größe und Mobilität sind sie sehr vielseitig einsetzbar und durch die steigende Rechenleistung dieser Geräte sind auch komplexere Anwendungsszenarien denkbar geworden. Für Mobiltelefone lassen sich einige Anforderungen zusammenfassen:

Vielseitigkeit Das Gerät muss diverse Dienste zur Verfügung stellen, zum Beispiel Basisdienste wie Telefonie und Nachrichtenaustausch, aber auch erweiterte Dienste wie Internetzugang oder GPS-Ortung.

Erweiterbarkeit Benutzer und Dienstleister sollten in der Lage sein, neue Dienste und Anwendungen für das Gerät zu erstellen und zu verbreiten.

Abstraktion Dem Benutzer sollen alle Funktionen zur Verfügung stehen, ohne dass dieser sich mit der Hardware oder der Implementierung des Systems auseinandersetzen muss.

Effizienz Die begrenzten Ressourcen eines Mobiltelefons erzwingen einen sehr effektiven Umgang mit den zur Verfügung stehenden Mitteln wie Arbeitsspeicher und Rechenleistung. Werden diese Ressourcen nicht optimal genutzt, kann kein performantes System bereitgestellt werden.

Sicherheit Durch die diversen, teils auch kommerziellen Einsatzgebiete der Mobiltelefone sind hohe Schutzmechanismen nötig. Diese reichen von der Abschottung der Daten gegenüber anderen Nutzern oder Prozessen über Verschlüsselung bis hin zu Trusted Computing.

In den folgenden Abschnitten wird der von Google entwickelte Open-Source Software-Stack für mobile Geräte vorgestellt. Dazu wird zuerst die Android-Architektur mit ihren verschiedenen Schichten beschrieben. Daraufhin werden die wichtigsten Komponenten der Architektur beleuchtet. Abschließend wird eine Einführung in das von Android bereitgestelle Applikations-Framework gegeben und die Kommunikationsarten zwischen Android-Prozessen beschrieben. Viele der Informationen zu Android sind aus dem Entwicklerhandbuch der Android-Entwickler-Homepage[1] entnommen [1].

[1] http://developer.android.com

3.1. Architektur

Das speziell für mobile Geräte entwickelte Android-Betriebssystem erfüllt die oben genannten Anforderungen. Die dabei genutzte Architektur wird in Abbildung 3.1 dargestellt.

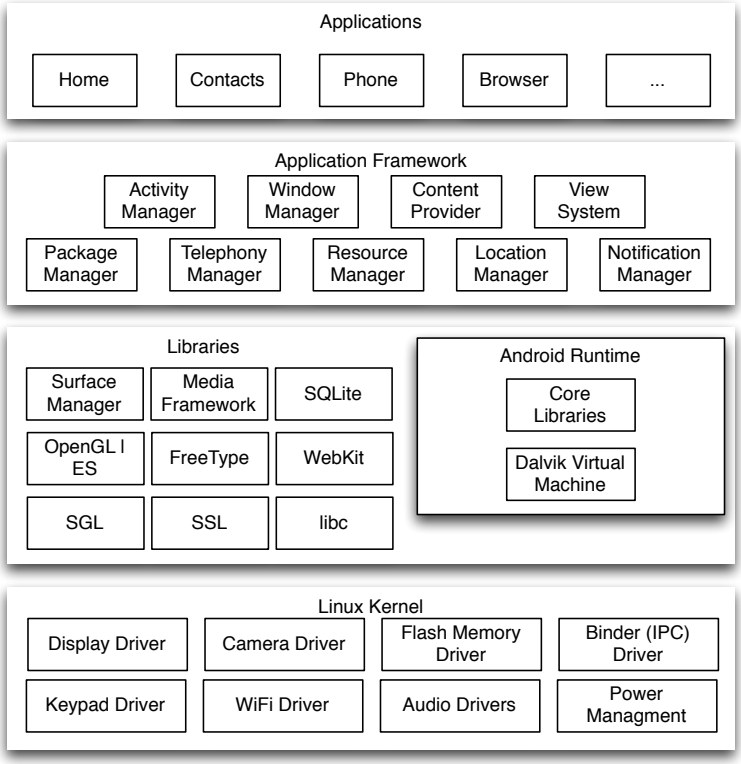

Abbildung 3.1.: Android System Architektur [1]

Sie besteht aus mehreren Schichten, welche jeweils eine definierte Aufgabe wahrnehmen.

3.1.1. Schichten

Die unterste Schicht bildet ein Linux Kernel, welcher Treiber und Services anbietet und so die Hardware **abstrahiert**. Die über dem Kernel liegenden Schichten nutzen beispielsweise Linux-Prozesse, um Anwendungen auszuführen. Diese Anwendungen profitieren von den im Kernel definierten Eigenschaften eines solchen Prozesses, wie beispielsweise die Benutzerverwaltung. Das Prozessmodell legt auch den Grundstein für das von Android verwendete **Sicherheitskonzept**, welches in Abschnitt 3.1.2 beschrieben ist.

Die nächste Schicht beinhaltet Libraries und die Android Runtime. Diese Android Runtime stellt modifizierte Java VMs zur Verfügung, in welchen die Anwendungen ausgeführt werden. Die Dalvik genannte virtuelle Maschine besitzt ein eigenes Dateiformat,

in welchem der Java Bytecode für eine Anwendung zusammengefasst wird. Dieses Dateiformat lässt sich mit .jar Archiven vergleichen, die im normalen Java-Umfeld mehrere Klassen zu einem Paket zusammenfassen. Dalvik executables (.dex) sind allerdings nicht komprimiert, sondern nur optimiert zusammengestellt. Der Speicherplatz wird so sehr **effizient** genutzt.

Das Application Framework stellt Applikationen wichtige Funktionen zur Verfügung. Es bietet Schnittstellen zur Darstellung von Inhalten, zum Zugriff auf das Dateisystem und auf andere Hardwareressourcen an. Weiterhin wird durch das Framework auch die Kommunikation zwischen den Anwendungen definiert. Durch dieses Framework lassen sich beliebige Anwendungen entwickeln und anschließend auf den mobilen Geräten nutzen. Dadurch wird die **Erweiterbarkeit** des Systems erhöht. Besonders wichtig ist hierbei, dass die Entwicklung nicht nur von dafür bestimmten Firmen übernommen werden kann, sondern dass das Framework für jeden zugänglich und Open-Source ist. Dadurch ist mit einer unbegrenzten Anzahl von Anwendungen zu rechnen.

Die große Anzahl möglicher Anwendungen, die auf der obersten Schicht angesiedelt werden, beschreibt die **Vielseitigkeit** des Systems. Auf dieser Ebene können Applikationen von Telefongesprächen bis zu Client Anwendungen für eine Unternehmenssoftware angesiedelt und entwickelt werden.

Dalvik Executables Wie im vorigen Abschnitt beschrieben, werden Java Anwendungen nicht als .jar-Archive, sondern als Dalvik executables ausgeliefert. Die Besonderheit der .dex Dateien liegt in der Zusammenfassung mehrerer .class-Dateien: Diese werden zuerst aufgesplittet. Die String-Konstanten werden für alle Klassen zusammengefasst, was viele Redundanzen verhindert. Statt in jeder Klasse je eine Tabelle aller Methoden-, Feld- und Klassennamen zu speichern, wird in einer .dex-Datei nur jeweils eine Tabelle gespeichert[2].

Einer der Entwickler der Dalvik VM, Dan Bornstein beschreibt in einem Video von der Google I/O 2008 [7] die Ersparnis durch dieses Verfahren. Er erklärt, dass durch diese Maßnahmen die unkomprimierte .dex-Datei eines Beispiels nur bis zu 44% der Größe einnimmt, welche die Java-Klassen zusammengenommen umfassen. Mit diesem Verfahren ergeben sich sogar kleinere Größen als bei komprimierten .jar-Dateien.

3.1.2. Sicherheitsmodell

Das wichtigste Sicherheitskonzept der Android-Plattform ist die Trennung der Applikationsprozesse. Dies wird unter anderem erreicht, indem jede Anwendung, die installiert wird, eine eigene, fortlaufende PID und einen eigenen User im System zugeordnet bekommt. Tabelle 3.1 zeigt einen Auszug der mit *ps* angezeigten Prozessliste. Daran sind die gestarteten Prozesse mit den ihnen zugeordneten Benutzernamen gut zu erkennen.

[2] http://dalvikvm.com

User	PID	NAME
root	1	/init
root	2	kthreadd
root	29	zygote
root	32	/system/bin/installd
keystore	33	/system/bin/keystore
root	35	/system/bin/sh
root	38	/sbin/adbd
radio	98	com.android.phone
app_1	101	android.process.acore
system	119	com.android.settings
app_4	135	com.android.calendar
app_18	150	com.android.alarmclock
app_21	224	de.fhhannover.inform.trust.tsse.test
app_20	230	de.fhhannover.inform.trust.tsse

Tabelle 3.1.: gekürzte Prozess Liste

Abgeschottete Ressourcen

Der Anwendung werden so ihre eigenen, abgeschotteten Ressourcen zugeordnet. Jeder Anwendung steht beispielsweise ein Pfad im Dateisystem zur Verfügung, auf den nur das Programm zugreifen kann. Dies wird in der gekürzten Tabelle 3.2 gezeigt. Diese Tabelle listet einen Teil der Anwendungs-Ordner in dem */data/data* Verzeichnis auf.

Rechte	User	Gruppe	Datum	Zeit	Name
drwxr-xr-x	app_20	app_20	2010-06-15	10:57	de.fhhannover.inform.trust.tsse
drwxr-xr-x	app_1	app_1	2010-06-15	10:55	com.android.inputmethod.latin
drwxr-xr-x	app_19	app_19	2010-06-15	10:56	com.android.mms
drwxr-xr-x	app_18	app_18	2010-06-15	10:56	com.android.alarmclock
drwxr-xr-x	app_17	app_17	2010-06-15	10:55	com.android.packageinstaller
drwxr-xr-x	app_4	app_4	2010-06-15	10:55	com.android.providers.calendar
drwxr-xr-x	app_1	app_1	2010-06-15	10:55	com.android.globalsearch
drwxr-xr-x	system	system	2010-06-15	10:55	com.android.server.vpn
drwxr-xr-x	app_5	app_5	2010-06-15	10:55	com.android.gallery
drwxr-xr-x	app_16	app_16	2010-06-15	10:55	com.android.certinstaller

Tabelle 3.2.: gekürzter Auszug des Dateisystems

Damit wird erreicht, dass ein manipuliertes Programm nur die eigenen Ressourcen auslesen und verändern kann, jedoch keinen Zugriff auf die Daten anderer Prozesse hat. Das Framework (siehe Abschnitt 3.2) stellt Methoden zur Verfügung, um trotz dieser Einschränkung Daten zwischen Prozessen auszutauschen.

Permissions

Eine weitere Sicherheitsmethode ist die Nutzung von Permissions. Eine Anwendung gibt dazu beispielsweise in der Manifest-Datei an, welche Dienste sie in Anspruch nimmt. Der Benutzer kann so vor der Installation sehen, was diese Anwendung für Dienste benutzt, und daraufhin entscheiden, ob er dieses zulassen möchte oder nicht. Wird durch eine Anwendung eine Aktion durchgeführt, die eine Permission benötigt, diese aber nicht im Manifest definiert, wird die Ausführung der Anwendung mit einer Exception abgebrochen.

3.2. Framework

Das Android Application Framework stellt die notwendigen Klassen und Schnittstellen bereit, damit Entwickler Anwendungen für die Android Plattform erstellen können. Das Framework bietet vor allem Methoden für den Zugriff auf die Ressourcen der Plattform und zur Definition von grafischen Benutzerschnittstellen. Mobile Geräte sind häufig mit vielen Eingabemöglichkeiten wie Touchpads, Beschleunigungssensoren und GPS ausgestattet. Diese Eingaben werden dem Anwendungsentwickler zur Verfügung gestellt.

Durch das Framework werden zwei wichtige Applikationstypen eingeführt: Activities und Services.

3.2.1. Activities

Eine Anwendung besteht aus einer oder mehreren Activities, welche vom Benutzer gestartet werden, ihm eine Benutzeroberfläche anbieten und auf seine Eingaben reagieren.

Dabei besitzt jede Activity einen genau definierten Lebenszyklus, der dem Entwickler ermöglicht, seine Anwendung ideal auf die Anforderungen eines mobilen Gerätes einzustellen. Besondere Anforderungen entstehen auf mobilen Geräten durch die geringeren Ressourcen dieser Geräte und durch mögliche Unterbrechungen einer Programmausführung. Wenn ein Telefongespräch auf dem mobilen Gerät ankommt, wird die laufende Anwendung pausiert, um das Gespräch zu ermöglichen. Die Anwendung muss deshalb dazu in der Lage sein, seinen aktuellen Stand zu sichern. Nach Beendigung des Telefonats wird die Activity fortgesetzt. Der gesicherte Stand muss daraufhin geladen werden, um dem Benutzer seinen aktuellen Programmfortschritt zu präsentieren. Um diese und andere Funktionen sicherzustellen, bietet die abstrakte Klasse Activity Methoden an, die der Entwickler dem Verhalten entsprechend überschreiben kann.

In einem normalen Ablauf einer Activity stellt diese verschiedene Benutzeroberflächen dar, in Android *View* genannt. Die Benutzerreaktion auf die Views kann von der Activity über Listener angenommen werden und steuert so den Programmablauf.

Neben den Java-Bestandteilen einer Activity, welche den Programmfluss steuern, existieren mehrere Konfigurationsdateien. Die wichtigste Konfigurationsdatei einer jeden Android-Anwendung ist die *Manifest.xml*. In dieser werden alle Komponenten eines Paketes definiert. Weiterhin enthält das Manifest die nötigen Permissions und Einstellungen, welche die Activities steuern. Zusätzliche Dateien beschreiben den Aufbau der

grafischen Benutzeroberfläche sowie Stringkonstanten, die in diesen Oberflächen genutzt werden.

Views erlauben eine Vielzahl von Komponenten, darunter Textfelder, Buttons und Checkboxen. Android bietet in seinem Software Development Kit (SDK) einen grafischen Editor an, mit dem diese erstellt und ergonomisch angeordnet werden können.

3.2.2. Services

Im Gegensatz zu Activities besitzen Services keine eigene Benutzeroberfläche. Ein Service bietet bestimmte Funktionen an, welche dann von Activities oder anderen Services genutzt werden können. Services besitzen einen Lebenszyklus, der etwas von dem einer Activity abweicht. Anstatt vom Benutzer aufgerufen zu werden, lassen sich Services nur von anderen Android-Komponenten aufgerufen. Dafür steht beispielsweise die Methode `onBind()` zur Verfügung, welche aufgerufen wird, um sich mit einem Service zu verbinden. Ein weiterer Unterschied besteht darin, dass ein Service durch seine Unabhängigkeit von einer grafischen Benutzeroberfläche nicht pausiert werden kann.

Die Funktionen, die der Service anbietet, werden über die Android Interface Definition Language (AIDL) spezifiziert. Eine AIDL-Datei, welche eine bestimmte Service-Schnittstelle definiert, kann mit dem zu Android gehörenden *aidl* Tool in Java Stub und Skeleton Klassen umgewandelt werden. `onBind()` liefert dieses Skeleton dann als Instanz eines *IBinder*-Objekts zurück.

Für den Transport von Daten können Java Basistypen verwendet werden. Sollen Objekte versandt werden, müssen diese das Interface *Parcelable* implementieren und eine AIDL muss angelegt werden.

3.3. Kommunikation zwischen Prozessen

Wie bereits beschrieben, ist es häufig nötig, zwischen verschiedenen Prozessen Informationen auszutauschen. Dafür bietet Android diverse Möglichkeiten.

Neben dem schon beschriebenen Aufruf eines Services, welcher die benötigten Daten als Rückgabewert liefert, ist es möglich, sich als Listener auf Events des Systems zu registrieren. Die dafür genutzten Komponenten werden `Broadcast Receivers` genannt. Aktivitäten und Dienste sind in der Lage, eigene Broadcasts abzusenden.

Broadcasts werden asynchron verschickt. Eine Anwendung muss dementsprechend auf sie reagieren können.

Laufzeitabhängige Aktionen des Gerätes könnten durch Broadcasts abgebildet werden. Dieses Verfahren eignet sich beispielsweise gut, um alle Applikationen über die Änderung einer Datei oder die Installation einer Anwendung zu informieren. Mit Broadcasts lassen sich nur schwer synchrone Anfragen abbilden. Für solche Fälle sind Services besser geeignet.

Eine weitere Art, um Informationen von anderen Prozessen zu erhalten, sind `Content Provider`. `Content Provider` stellen Daten für andere Prozesse zur Verfügung. Die in Abschnitt 3.1.2 beschriebene Abschottung der Prozesse untereinander hat zur Folge, dass kein gemeinsamer Speicher auf dem Gerät existiert. Will eine Anwendung also Daten exportieren, muss sie einen `Content Provider` registrieren. Um an diese Daten zu gelangen, kann ein anderer Prozess dann einen `Content Resolver` nutzen. Dieser liefert die Daten und abstrahiert dabei von den Kommunikationmechanismen der beiden involvierten Prozesse.

3.3.1. Intents

Für die Kommunikation zwischen Prozessen wird ein *Intent* benötigt. Arno Becker und Marcus Pant beschreiben Intents wie folgt [5]:

> Intents verbinden unabhängige Komponenten, also Activities, Services, Content Provider oder Broadcast Receiver, untereinander zu einem Gesamtsystem und schaffen eine Verbindung zur Android-Plattform.

Mit einem Intent wird also ein Verbindungswunsch beschreiben. Ein Intent besitzt verschiedene Attribute, die bestimmen, was durch das Intent ausgedrückt werden soll. Um innerhalb von Anwendungen eine Activity oder einen Service zu starten, lassen sich explizite Intents verwenden. Diese enthalten den Namen der Klasse, an die sie ausgeliefert werden sollen. Da dies in vielen Fällen nicht möglich ist, zum Beispiel bei dem Aufruf einer Klasse von anderen Entwicklern, ist dieser Name optional. Ist er nicht angegeben, handelt es sich um implizite Intents.

Bei impliziten Intents ist die Auslieferung an eine bestimmte Klasse nicht möglich. Sie werden also ohne Empfänger verschickt. Das Intent enthält die Art der Aktion, die ausgeführt werden soll. Weiterhin ist es möglich, eine Kategorie der Aktion zu bestimmen und zusätzliche Attribute an das Intent anzuhängen. Mit dem Intent wird über die Aktion und die, wie Parameter der Aktion wirkenden, Attribute dem Empfänger mitgeteilt, was ausgeführt werden soll.

Diese Informationen werden mit denen der Manifest-Dateien der vorhandenen Anwendungen abgeglichen. Im Manifest kann für jede Activity, jeden Service und jeden BroadcastReceiver ein `<Intent-Filter>` Block spezifiziert werden. In diesem werden die Aktion, die Kategorie und noch weitere Attribute des Dienstes definiert, um implizite Intents damit zu vergleichen. Passt ein Intent zu den im Intent-Filter definierten Einschränkungen, wird es ausgeliefert.

Der Service oder die Activity, die durch das Intent gestartet wurde, sind in der Lage, die im Intent vorhandenen Informationen zu nutzen und entsprechend darauf zu reagieren.

Ein beispielhafter Ablauf einer Verbindung zwischen einer Activity und einem Service wird in Abbildung 3.2 dargestellt. Um sich mit dem Service zu verbinden, wird ein Intent-Objekt erzeugt, welches mit der durchzuführenden Aktion initialisiert wird. Weiterhin wird eine *ServiceConnection* erzeugt, welche die Verbindung zu dem Service behandelt. Dieses Objekt bietet zwei Methoden, die von dem Framework aufgerufen

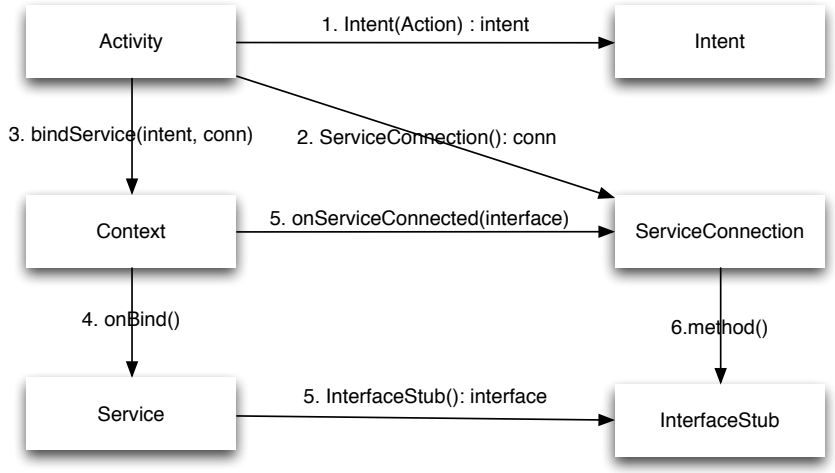

Abbildung 3.2.: Ablauf einer Service-Verbindung

werden. Mit `onServiceConnected` kann der Verbindungsaufbau mit dem Service behandelt werden. `onServiceDisconnected` wird aufgerufen, wenn die Verbindung abgebaut wird.

Um mit dem Service zu kommunizieren, werden diese Methoden von der aufrufenden Anwendung implementiert. Mitsamt des Intents wird das Objekt über die Methode `bindService` an den Anwendungskontext übergeben. Über diesen Kontext wird der Service gestartet und auf diesem die `onBind`-Methode aufgerufen. Die Methode besitzt als Rückgabewert ein Interface. Der Service liefert die Serviceimplementierung zurück, welche das Interface implementiert. Die Implementierung wird wiederum an die `ServiceConnection` weitergeleitet. Die Connection kann anschließend die notwendigen Aufrufe auf dem Service über das Interface durchführen.

3.3.2. Android Interface Definition Language

Wie bei den Android Services (3.2.2) beschrieben, wird für die Interprozesskommunikation von Android die AIDL verwendet. Mit dieser werden wie bei der CORBA Interface Definition Language die Schnittstellen der Dienste beschrieben, welche aufgerufen werden können. Sowohl der Server als auch der Client, die über diese Schnittstelle kommunizieren, generieren Code aus der AIDL. Dieser Code ist für die Übertragung der Daten, die ausgetauscht werden sollen, zuständig.

Auf der Serverseite wird der durch die Schnittstellenbeschreibung generierte Code abgeleitet und als Interface des Dienstes implementiert. Ein durch ein Intent gestarteter Service ist in der Lage, in seiner Lebenszyklusmethode `onBind()` diese Implementierung seines Interfaces an den Aufrufer zurückzugeben. Dieser besitzt eine AIDL-Beschreibung dieses Interfaces und hat durch das *aidl*-Werkzeug den Code für die Kommunikation generiert. Daher ist er in der Lage, auf die Methoden des Dienstes zuzugreifen, ohne sich mit der Übertragung zu befassen.

Die beschriebenen Zusammenhänge werden in Abbildung 3.3 gezeigt. Für den Client und den Server werden aus der AIDL-Beschreibung der Code für die Übertragung und das Interface erzeugt. Diese können die daraus entstehenden Klassen nutzen, um miteinander zu kommunizieren.

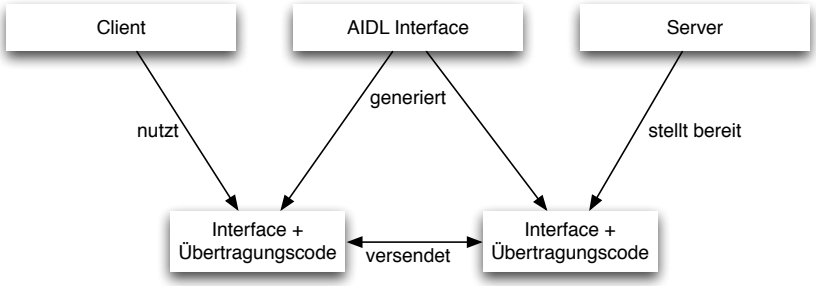

Abbildung 3.3.: Beispiel der Nutzung des AIDL-Interfaces

3.4. Messungen

Die in den Trusted Computing Grundlagen definierte Art um Komponenten eines Systems zu identifizieren, ist die Messung der Komponenten (Siehe Abschnitt 2.2). Wie beschrieben wird für diese Messung der Hashwert einer binären Datei verwendet. Dieser Abschnitt untersucht die in der Android Architektur verwendeten Komponenten auf ihre Messbarkeit hin.

3.4.1. Kernel

Der Kernel liegt als Image in binärer Form im Dateisystem vor und wird im regulären Betrieb nur durch Updates geändert. Er selbst wird während der Laufzeit der Plattform nicht modifiziert. Eine Messung sollte in einer manipulationsfreien Umgebung also immer denselben Wert liefern.

Da sich die Werte nicht ändern, kann einer bestimmten Kernel-Version ein spezifischer Wert zugewiesen werden. Der während dieser Arbeit für den Android-Emulator erstellte Kernel mit der Version 2.6.29 und eigenen Modifikationen besitzt einen eindeutigen Hashwert:

```
927b3243e044b462292f29d22bf92c4cae9336d5  kernel-qemu
```

3.4.2. Libraries

Neben dem Kernel bestimmen die Bibliotheken maßgeblich das Verhalten des Systems. Da sich die Libraries ebenso wie der Kernel in binärer Form auf dem System befinden und nur durch Updates des Systems geändert werden, ist eine Messung möglich.

3.4.3. Dalvik

Auch wenn sich die Dalvik VM auf der selben Ebene befindet wie die oben beschriebenen Libraries, sollte Dalvik gesondert betrachtet werden. Alle Anwendungen, die die Benutzer starten, werden von Dalvik bearbeitet. Das bedeutet, dass es zwingend erforderlich ist, die Identität der VM zu kennen, um Rückschlüsse auf die Vertrauenswürdigkeit der gestarteten Anwendungen ziehen zu können. Bei einem solchen Start lädt der Dalvik-Hauptprozess das installierte Android-Package (.apk) und somit die darin befindlichen Konfigurationsdateien sowie die Dalvik-Executables (.dex). Um die Trennung der Ressourcen zu gewährleisten und jede Applikation in ihrem eigenen Prozess und ihrer eigenen Virtuellen Maschine zu starten, wird anschließend ein `fork()` ausgeführt.

Wie die oben genannten Libraries liegt die Dalvik VM auch als Bibliothek vor und kann daher ebenfalls gemessen werden.

3.4.4. Dalvik Executables

Anwendungen werden als Android-Packages in dem Android Software Stack auf der obersten Ebene installiert. Diese Packages enthalten die Dalvik-Executables, welche den Java-Bytecode für die Plattform enthalten. Weiterhin sind in diesen Packages die Konfigurationsdateien der Anwendung, wie zum Beispiel das Manifest, enthalten.

Die von Android genutzten Dalvik-Executables werden bei der Ausführung einer Applikation nicht modifiziert. Dadurch ist der Hashwert der Datei vor und nach Ausführung der Anwendung identisch. Die Konfigurationsdateien werden auch nicht durch die Ausführung modifiziert. Die Daten der Applikation werden im Dateisystem oder in der von Android zur Verfügung gestellten Datenbank abgelegt, so dass sie die .dex-File nicht verändern. Insgesamt beinhaltet ein Android-Package also nur statische Komponenten. So kann durch eine Messung des Android-Packages die Identität der Anwendung festgestellt werden.

Da sich durch Erweiterungen der Applikation auch die .dex-File ändert, lassen sich auch kleinste Versionsänderungen und Manipulationen identifizieren. Dies ist gewollt um mögliche Manipulationen zu erkennen.

4. Lösungsansätze

Wie in Kapitel 3 beschrieben, sind mobile Geräte speziellen Bedrohungen ausgesetzt, die bei konventionellen Systemen nicht vorliegen. Diesen Bedrohungen kann auf modernen mobilen Geräten entgegengewirkt werden, da diese wie in Abschnitt 3.1.1 beschrieben um verschiedene Komponenten und Anwendungen erweitert werden können. Dazu eignet sich Trusted Computing besonders, da es den beteiligten Parteien eine Einschätzung zur Sicherheit der Plattform ermöglicht.

Verschiedene Parteien sind bei einem mobilen Gerät involviert. Anhand eines Beispiels lassen sich diese Parteien identifizieren:

Ein Versicherungskonzern, welcher seinen Außendienstmitarbeitern ein komfortableres Arbeiten ermöglichen möchte, hat diese mit mobilen Telefonen ausgerüstet. Über eine darauf installierte Anwendung können Versicherungsverträge gebucht werden. Diese Anwendung überträgt die Kundendaten an einen Server der Firma und führt eine Prüfung der Bonität der Kunden bei einer Bank durch.

Der Versicherungskonzern ist der Eigentümer der mobilen Geräte. Ihm ist besonders wichtig, dass die Geräte in seinem Eigentum verbleiben und nur für Zwecke genutzt werden, zu denen sie vorgesehen sind. Der Außendienstmitarbeiter ist Benutzer der Plattform. Als Benutzer ist ihm die Sicherheit seiner Daten, der Schutz vor Manipulation der genutzten Programme und der Schutz seiner Privatsphäre wichtig. Weiterhin ist er an einer sicheren Kommunikation mit seinen Service Providern interessiert. Service Provider sind in dem Beispiel sowohl die Bank als auch der Versicherungskonzern. Als Erbringer eines Dienstes ist eine Authentisierung und eine Autorisierung des Benutzers nötig sowie ein sicherer Kommunikationskanal, welcher die Integrität und Vertraulichkeit der übertragenden Daten gewährleistet. Zusätzlich wäre eine Aussage über den Zustand des anfragenden Clients ein für den Service Provider wünschenswertes Sicherheitskriterium. Ein weiterer Beteiligter ist der Anwendungsentwickler, welcher sichergehen möchte, dass sein geistiges Eigentum nicht verändert wird und der darauf bedacht ist, dass nur bezahlende Kunden seine Software nutzen können. Der Kunde kommt zwar nicht direkt mit dem mobilen Gerät in Kontakt, dennoch werden seine Daten auf diesem Gerät abgelegt und übertragen. Daher ist auch er an der Sicherheit seiner Daten und seiner Privatsphäre interessiert.

Dieses Kapitel definiert Anforderungen an ein System, das den Schutz der beteiligten Personen gewährleisten soll, und kombiniert Lösungsansätze, welche genutzt werden können, um diesen Anforderungen zu entsprechen.

4.1. Anforderungen

Aus dem beschriebenen Anwendungsfallbeispiel dieses Kapitels ergibt sich eine Zahl von Forderungen an eine Integrationslösung. Diese sind in der folgenden Liste zusammengefasst:

1. Schutz des physischen Eigentums
2. Sicherstellen der Integrität und Vertraulichkeit der Daten auf den mobilen Geräten
3. Sicherung der Privatsphäre der Benutzer
4. Sicherstellen von Authentisierung und Autorisierung der Plattform gegenüber Dienstleistern
5. Vertrauensentscheidungen der Benutzer erlauben
6. Sichere Übertragungen etablieren

Im Folgenden werden Lösungsansätze vorgestellt, welche die Anforderungen der beteiligten Parteien durch das Nutzen von TrustedComputingTechnologien abbilden.

4.2. Secure und Authenticated Boot

Ein Angreifer, welcher sich Zugriff auf die Plattform beschafft hat, ist in der Lage, Manipulationen an den Komponenten der Plattform vorzunehmen. Er tauscht beispielsweise den Kernel gegen einen Kernel aus, der sämtliche Prozesse überwacht und alle Informationen aufbereitet an einen Server sendet. Er erhofft sich dadurch sensible Passwörter für Bank- und Firmenaccounts auszuspähen. Nach Durchführung dieser Manipulation startet der Benutzer sein System. Es ist für ihn nur schwer möglich, die Manipulation zu entdecken.

Der für den Eigentümer und den Benutzer wichtige Aspekt, dass das System nicht manipuliert werden kann, wird durch einen Secure oder Authenticated Boot erreicht. Diese beiden Mechanismen überprüfen während des Boot-Prozesses die gestarteten Komponenten und ermöglichen damit eine Entscheidung, ob diesen vertraut werden kann oder nicht. Wenn Secure Boot aktiviert ist, wird der Boot unterbrochen sobald eine Manipulation entdeckt wird und macht somit die Manipulation des Kernels offensichtlich. Dadurch wird Anforderung (5) erfüllt. Der Außendienstmitarbeiter kann durch dieses Verfahren sicherstellen, dass er seine Plattform nur dann starten kann, wenn der Status des Systems vertrauenswürdig ist.

4.2.1. Funktion

Secure Boot wird von Löhr et al. als ein Pattern für Trusted Computing definiert [13]. Secure Boot startet das System nur dann, wenn alle Komponenten vertrauenswürdig sind. Dazu wird der Boot-Prozess so modifiziert, dass nur Komponenten gestartet werden, welche gemessen und verifiziert wurden. Secure Boot verhindert die Nutzung des Gerätes, wenn es manipuliert wurde und erfüllt somit die Anforderung (1). Die Messungen werden hierbei durch eine Chain of Trust (siehe Abschnitt 2.5) ausgeführt.

Dietrich und Winter beschreiben eine Architektur, mit der Secure Boot auf mobilen Plattformen mit einem Mobile Trusted Module (MTM) unterstützt werden kann [9]. Dazu zeigen sie die Integration eines Secure-Boot-Prozesses.

In einer abgewandelten Form existiert dieses Verfahren unter dem Namen Authenticated Boot. Bei diesem Verfahren wird während des Bootvorgangs auch jede Komponente gemessen. Das System wird in diesem Verfahren auf jeden Fall gestartet und eine Verifikation der Werte wird erst später durchgeführt. Ergänzend zum Authenticated Boot-Verfahren verhindert Secure Boot, dass ein nicht vertrauenswürdiges System gestartet wird. In beiden Verfahren ist es für den Benutzer möglich, zu erkennen ob das System manipuliert wurde. Diese Verfahren betrachten dabei immer nur den Start eines Programmes um die Verifikation durchzuführen. Ein Angriff während der Laufzeit bleibt bis zu dem nächsten Start unentdeckt.

4.2.2. Implementierungsbeispiel

Trusted Grub[1] ist ein Beispiel für einen Bootloader, mit Authenticated Boot Implementierung. Dieser ist so definiert, dass es sich selbst und die steuernde Konfigurationsdatei misst. Anschließend werden alle in der Konfiguration definierten Komponenten gemessen und gestartet. Die in den PCRs gespeicherten Werte werden bei jedem Boot neu berechnet.

Um Secure Boot etablieren zu können, ist ein Referenzwert für alle gemessenen Werte nötig. Für diesen Fall bietet Trusted Grub eine Option an, um dies zu aktivieren. Es kann eine Datei definiert werden, welche den entsprechenden Referenzwert hält. Nur wenn dieser Wert mit dem gemessenen übereinstimmt, wird das System gestartet.

4.3. Remote Attestation

Um eine vertrauenswürdige Verbindung zwischen einem Dienstleister und einem Benutzer herzustellen, ist es nötig, Informationen für beide Seiten bereitzustellen. Es ist zum Beispiel wichtig, dass jeder Partner identifiziert werden kann, um zu entscheiden, ob es sich um den gewünschten Dienstleister handelt und um eine Autorisierung zu ermöglichen. Es ist wichtig festzustellen, ob der Benutzer berechtigt ist, bestimmte Operationen durchzuführen. Remote Attestation ermöglicht es weiterhin Service Providern, eine Entscheidung über den Zustand einer Benutzer-Plattform zu treffen. Wenn der Zustand der

[1] http://sourceforge.net/projects/trustedgrub/

Plattform und die Identifizierung des Benutzers gültig sind, kann eine Benutzung erlaubt werden.

Das klassische Anwendungsbeispiel für die Remote Attestation ist die Zugangsberechtigung zu einem lokalen Netzwerk anhand einer Vertrauensentscheidung. Dieses wird beispielsweise durch Trusted Network Connect abgebildet.

Ein weiterer Anwendungsfall geht über das lokale Netzwerk hinaus. Der Benutzer möchte mit seinem System den Dienst einer Bank verwenden um beispielsweise die Bonität eines Kunden zu prüfen. Dazu baut er eine Verbindung zu der Bank auf, welche daraufhin den Benutzer authentisieren möchte. Dies kann mit einer Angabe eines Benutzernamens und eines Passworts gelöst werden. Um die Sicherheit der Transaktionen sicherzustellen, kann die Bank zusätzlich eine Verifizierung der Plattform des Benutzers anfordern. Dazu wird eine Remote Attestation der Plattform durchgeführt und anhand der Policy der Bank überprüft. Weiterhin kann durch den Aufbau einer vertrauenswürdigen Verbindung die Integrität und Vertraulichkeit der Daten sichergestellt werden. Dies erfüllt Anforderung (6) nach einer sicheren Übertragung.

Weiterhin entkoppelt sie die Entscheidung, welche Konfigurationen vertrauenswürdig sind, von den einzelnen Plattformen und Nutzern eines Netzwerks. Diese können und müssen nicht wissen was vertrauenswürdig ist, da die Entscheidung darüber entfernt getroffen wird.

Als Beispiel hierfür kann der oben genannte Versicherungskonzern gesehen werden, welcher Sachbearbeiter angestellt hat, die mit einem festen Arbeitsplatz ausgestattet sind. Diese Benutzer verfügen nicht über technisches Fachwissen, um entscheiden zu können, welche Version eines Programmes vertrauenswürdig ist und welche nicht. Der Administrator des Netzwerkes hingegen besitzt mehr Übersicht über bekannte Schwachstellen und Programmversionen und ist in der Lage, Policies zu definieren, anhand derer Vertrauensentscheidungen netzwerkweit durchgeführt werden können. Diese Aufgabe ist schon für einen Administrator mit sehr viel Aufwand verbunden, so dass es die Möglichkeiten eines Sachbearbeiters weit übersteigt.

4.3.1. Funktion

Um bestimmte Dienste in Anspruch nehmen zu können, müssen diese Dienste entscheiden können, ob der Plattform des Clients vertraut werden kann. Auch für den Client kann es wichtig sein Informationen über die Vertrauenswürdigkeit des Dienstanbieters zu erhalten. Da es sich jedoch in den meisten Fällen bei dem Client um eine Privatperson und bei dem Dienstanbieter um Institutionen wie eine Bank oder ein Unternehmen handelt, sind die entsprechenden Schutzmechanismen sehr unterschiedlich. Es wird angenommen, dass bei einem Dienstanbieter stark geschützte Server eingesetzt werden, wohingegen der Client eher schwach gesicherte Rechner einsetzt. Daher wird in den meisten Fällen die Überprüfung des Clients angestrebt, wobei auch die andere Richtung möglich ist. Für diese Überprüfung werden durch die Nutzung einer Chain of Trust alle Komponenten gemessen und einem Verifier bereitgestellt. Dieser prüft die gemessene Kette und trifft aufgrund seiner Policy eine Entscheidung. Eine solche Policy definiert zum Beispiel, welchen Programmen in welchen Versionen vertraut werden soll.

Die Platform Configuration Register werden signiert und dem Verifier zusammen mit dem Stored Measurement Log übermittelt. Wie in Abschnitt 2.4 beschrieben, werden die PCRs mit wechselnden AIKs signiert, um gegen Angriffe geschützt zu sein und gleichzeitig die Privatsphäre der Plattform und des Benutzers nicht zu gefährden. Dies erfüllt Anforderung (3). Der Benutzer kann somit sicher sein, dass seine Aktionen nicht gespeichert oder korreliert werden können.

Der Verifier kann aus dem SML errechnen, ob die gesandten PCR-Werte korrekt sind. Ergibt die Berechnung keine Übereinstimmung, ist eine Manipulation von entweder dem SML oder den Komponenten aufgetreten, deren Messungen in dem PCR gespeichert wurden. In diesen Fällen kann keine positive Vertrauensentscheidung gefällt werden. Zusätzlich werden die im SML angegebenen Programme und Versionen mit der Policy abgeglichen. Sollte eine Konfiguration vorhanden sein, welche einen sicheren Betrieb ausschließt, wird die Vertrauensentscheidung auch negativ ausfallen.

4.4. Local Verification

Der Benutzer der mobilen Plattform installiert viele Anwendungen. Für ihn ist wichtig, dass eine Manipulation der Anwendungen nicht möglich ist. Ein Angreifer, welcher sich Zugang zu der Plattform des Benutzers verschafft hat, kann verschiedene Angriffe durchführen. Er könnte beispielsweise eigene Applikationen auf das Gerät übertragen oder bestehende Applikationen verändern, so dass diese seinen Bedürfnissen entsprechen. Die lokale Verifizierung versucht durch die Aufnahme eines Normalzustands des Systems sicherzustellen, dass die Angriffe unterbunden oder erkannt werden können.

Auf mobilen Geräten kann eine dauerhafte Verbindung zu anderen Geräten nicht vorausgesetzt werden. Dies birgt Probleme, zum Beispiel bei Remote Attestation. Bei der Überprüfung, ob eine Anwendung ausgeführt werden darf (oder nicht), wäre ein Remote Verifier nicht sinnvoll. Dem Benutzer wäre es nicht möglich, Applikationen zu starten, wenn er sich in einem Bereich ohne Netzzugriff aufhält. Daher muss für diesen Anwendungsfall eine lokale Verifizierung ermöglicht werden.

Allerdings ist die Funktion dieses Verfahrens sehr von dem Schutz der, die Verifizierung durchführenden, Komponenten abhängig. Wird die Metrik oder der Verifizierer manipuliert, kann nicht mehr sichergestellt werden, dass die Anforderungen erfüllt werden. Sobald ein Teil des lokalen Verifizierungsmechanismus kompromittiert wurde, kann keiner Aussage des Systems mehr Vertraut werden. Eine Remote Attestation würde jedoch zu einer Aufdeckung dieser Manipulation führen. Beide Verfahren ergänzen sich also in der Betrachtung des Sicherheitsstatus der Systems. Durch Schutzmaßnahmen, die auf die Metrik und den Verifizierer angewendet werden, lässt sich die Gefahr reduzieren.

Obwohl die lokale Verifizierung Angriffspunkte bietet, wird das System durch die Überprüfung der gestarteten Anwendungen sicherer. Wie bei dem Konzept aus der Spezifikation der Mobil Phone Working Group [25] ist allerdings Vertrauen in eine Root of Trust nötig. Je näher an der Hardware diese Root of Trust eingesetzt werden kann, desto sicherer wird das System.

4.4.1. Funktion

Um die lokale Verifikation zu realisieren, wird eine lokale Metrik benötigt. Diese Metrik bestimmt, welchen Anwendungen vertraut wird. Dazu wird zu jeder Anwendung der Hashwert der ausführbaren Binary gespeichert. Eine solche Metrik kann erzeugt werden, indem alle vom Benutzer installierten Anwendungen gemessen werden. Der Installationsprozess sollte also für die Pflege der Metrik verantwortlich sein.

Die wichtigste Komponente der lokalen Verifizierung ist dafür zuständig, vor dem Start der Anwendung eine weitere Messung durchzuführen und diese anschließend mit der Metrik zu vergleichen. Da sich im Normalbetrieb solche Anwendungen nicht ändern, kann der Messwert beim Start mit dem Installationswert verglichen werden. Sind diese Werte identisch, ist es durch die Eigenschaften einer Hashfunktion fast sicher, dass es sich um die identische Binary handelt. Solche Binaries dürfen gestartet werden. Befindet sich zu einer Anwendung kein Messwert in der Metrik, oder besitzt er einen anderen Messwert, wird kein Start durchgeführt und dies dem Benutzer mitgeteilt.

So kann dieses Verfahren den Benutzer des Gerätes davor schützen, dass keine Anwendungen auf anderen Wegen als dem Installationsprozess auf das Gerät installiert werden. Weiterhin verhindert es, dass Anwendungen, welche durch etwaige Manipulationen von anderen Prozessen oder Personen verändert wurden, gestartet werden. Dies erfüllt Anforderung (2). Für den Versicherungskonzern bedeutet dies, dass die Daten, welche der Kunde dem Versicherungsmitarbeiter anvertraut, nicht durch kompromittierte Anwendungen verändert werden können.

Es kann allerdings nicht die Installation von Schadprogrammen verhindern. Wenn der Benutzer ein solches Programm aus einem Repository lädt, wird es in der Metrik gespeichert und sie wird beim Start des Programmes nicht erkennen, dass es sich um ein Schadprogramm handeln könnte. Man könnte allerdings eine weitere Sicherheitsmaßnahme in den Installationsprozess integrieren. Dieser könnte aus den Anwendungsrepositories mit einer Blacklist von Anwendungen versorgt werden, welche nicht in die Metrik aufgenommen beziehungsweise aus der Metrik entfernt werden sollen. So ließe sich ein Starten einer Anwendung verhindern, nachdem die schädliche Wirkung bekannt geworden ist. Dieses Verfahren benötigt allerdings der Zustimmung des Nutzers für Änderungen, um nicht zuviel Kontrolle über das Gerät an ein solches Repository abzugeben. Sollte ein Unternehmen ein eigenes Repository betreiben liesse sich auch eine Whitelist der erlaubte Anwendungen realisieren. In einem solchen Fall ist auch de Kontrolle des Repositories über die Geräte ein gewünschtes Verhalten.

Dies würde bereits bestehende Schutzmechanismen von Seiten des von Android betriebenen Repositories erweitern, welches zum Beispiel nur signierte Software installiert und weiterhin in der Lage ist, bestimmte Software von dem Gerät zu löschen und zu installieren [17].

4.5. Zusammenfassung der Anforderungen

Die in diesem Kapitel gezeigten Anforderungen sind sehr vielfältig und aus der Sicht verschiedener Gruppen unterschiedlich wichtig. Daher werden sie noch einmal zusammengefasst und bewertet.

Anforderung	Lösungsansatz	Bewertung
Schutz des Eigentums	Secure Boot	Durch die Unterbrechung des Bootvorgangs kann ein manipuliertes System nicht gestartet werden
Sicherstellen der Integrität der Daten	Lokale Verifizierung	Macht Angriffe sichtbar und schützt so vor ihnen
Sicherung der Privatsphäre	Remote Attestation	Besonders wichtig für die Akzeptanz durch den Benutzer
Sicherstellen von Authentisierung und Autorisierung	Remote Attestation	Lässt Dienste nur durch befugte Personen ausführen
Vertrauensentscheidungen erlauben	Secure & Authenticated Boot	Notwendig für den Benutzer um ein manipuliertes System früh zu erkennen
Sichere Übertragungen etablieren	Remote Attestation	Abhängig von dem Sicherungsbedürfnis der Kommunikation

Tabelle 4.1.: Zusammenfassung der Anforderungen

Die betrachteten Lösungsansätze ergeben eine Gesamtlösung, welche die beschriebenen Anforderungen erfüllt. Für ein sicheres Gesamtkonzept sollten alle Teile zusammen genutzt werden, da sie teilweise aufeinander aufbauen und voneinander profitieren. Der hier vorgestellte lokale Verifizierer bietet zum Beispiel ohne zusätzliche Sicherheitsmaßnahmen der Metrik keinen Schutz der Anwendungen. Er bedarf der Sicherung über Maßnahmen wie Secure Boot, damit eine Erhöhung der Sicherheit erreicht werden kann.

5. Konzept zur Integration von Trusted Computing Funktionen in Android

Um die genannten Lösungsansätze auf einer mobilen Plattform realisieren zu können, müssen verschiedene Komponenten auf dem Gerät verfügbar sein.

Secure und Authenticated Boot setzen eine komplette Chain of Trust und damit ein TPM voraus. Da im Moment keine TPM-Hardware für mobile Geräte vorhanden ist und der Bootprozess geräte- und herstellerspezifisch ist, können diese Verfahren im Laufe dieser Arbeit nicht weiter betrachtet werden. Da allerdings zu erwarten ist, dass diese Unterstützung bereitgestellt wird, kann dieser Ansatz nachgelagert hinzugefügt werden, um das Konzept zu vervollständigen.

Remote Attestation nutzt neben einer Chain of Trust auch einen Prozess, welcher die Messwerte aus dem TPM ausliest und an den Verifier übermittelt. Für solche Zwecke kann beispielsweise ein TNC-Client verwendet werden, der ein Attestation IMC/IMV-Paar besitzt. Da die Schnittstelle des TPM nicht von jeder Applikation selbst implementiert werden soll, wird ein so genannter TCG Software Stack benötigt. Über diesen wird der Zugriff für die Remote Attestation realisiert.

Die von der lokalen Verifizierung benötigten Komponenten unterscheiden sich von denen, die von Secure Boot und Remote Attestation verwendet werden. Local Verification nutzt nicht direkt die Vorteile von Trusted Computing. Es werden Messungen durchgeführt, die drei spezielle Teile benötigen. Komponente eins ist dafür zuständig alle installierten Programme zu messen und zu speichern. Die zweite Komponente wird für das ablegen dieser Messungen genutzt. Der letzte benötigte Baustein ist dafür verantwortlich, die in der Metrik gespeicherten Informationen auszuwerten und die daraus entstehenden Regeln durchzusetzen.

Um die in den Lösungsansätzen genannten Anforderungen für die Plattform verfügbar zu machen, müssen die oben genannten Komponenten der Ansätze Secure und Authenticated Boot, Remote Attestation sowie local Verification auf ihr etabliert werden. Daraus ergeben sich die folgenden sechs Komponenten, welche auf der mobilen Plattform verfügbar gemacht werden sollen.

1. Chain of Trust
2. TCG Software Stack
3. Remote Attestation Prozess
4. Angepasster Installationsprozess
5. Prozess zur Durchsetzung der sich aus der Metrik ergebenden Regeln
6. Metrik

Wenn diese Komponenten in vollem Umfang auf dem Gerät zur Verfügung stehen, können alle beschriebenen Anforderungen erfüllt werden. In diesem Kapitel werden verschiedene Möglichkeiten der Integration für diese Komponenten abgewogen und anschließend wird die konzeptionelle Einbettung beschrieben. Im Abschnitt 5.7 wird schließlich gezeigt, wie die ausgewählten Komponenten in die Android-Plattform integriert werden können.

5.1. Chain of Trust

Um die in den Grundlagen beschriebene Chain of Trust(2.5) abbilden zu können, ist es nötig, eine Root of Trust for Measurement zu erstellen. Sie stellt einige Anforderungen an das System, die für die Funktion notwendig sind. Die RTM muss sehr früh im Bootprozess verfügbar sein, um die zu bootenden Komponenten zu messen. Weiterhin sollte sie so angelegt werden, dass eine Manipulation nicht möglich ist. In Plattformen mit einem Trusted Platform Module wird dafür häufig ein Bereich des BIOS verwendet, welcher den Bootloader misst, bevor dieser ausgeführt wird. Dies ist im Fall der verwendeten, emulierten Hardware und ohne Herstellerunterstützung nicht möglich. Daher muss auf andere Komponenten zurückgegriffen und Einschränkungen hingenommen werden. Im folgenden muss zwischen zwei Positionen unterschieden werden. Es ist davon auszugehen, dass in Zukunft Geräte mit TPM oder MTM Hardware ausgeliefert werden. Auf diesen Geräten sind die genannten Konzepte vollständig anzuwenden. Für den hier vorgestellten Prototypen muss allerdings ein Hardwareemulator genutzt werden, mitsamt einem emulierten TPM. Daher werden die nötigen einschnitte innerhalb der Konzepte und der Implementierung dahingehend beschrieben.

Mögliche Komponenten für die Positionierung der Root of Trust for Measurement sind der Bootloader sowie der Kernel. Je früher die RTM in den Bootprozess eingreifen kann, desto mehr Komponenten lassen sich messen.

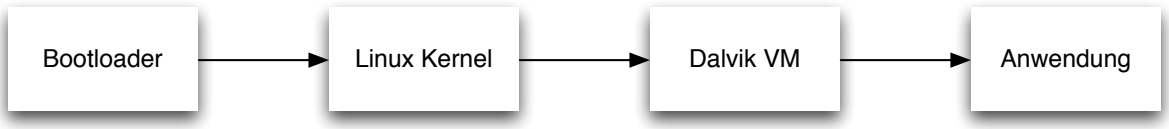

Abbildung 5.1.: Der Bootprozess

Bild 5.1 zeigt die wichtigsten Komponenten des Bootprozesses.

Am besten geeignet ist der Bootloader, welcher früh im System aktiv wird und so eine Messung aller anderen Komponenten ermöglicht. Die durchgeführte Messung des später gestarteten Kernels muss allerdings sicher abgelegt werden, damit später auch auf diesen Wert zugegriffen werden kann. Daher muss eine Implementierung im Bootloader sowohl einen Mechanismus zur Messung als auch zur sicheren Ablage zur Verfügung stellen. Dies ist problematisch, da solche Mechanismen nur durch die Hardware eines Systems bereitgestellt werden können. Aktuelle Smartphones verfügen momentan nicht über diese Hardwareunterstützung, was Messungen vor dem Kernel verhindert.

Bei einer Implementierung im Kernel des Systems sind diese Mechanismen einfacher zu realisieren. Eine Messung der Dalvik VM sowie aller von Benutzern des Systems installierten Anwendungen ist möglich. Allerdings lässt sich der Kernel nicht sicher messen, was einen deutlichen Informationsverlust in Bezug auf die Aussage, ob das System verändert wurde oder nicht, bedeutet. Sollte der Kernel sich selbst messen, kann diesem Messwert nicht vertraut werden, da ein kompromittierter Kernel eine solche Messung dahingehend beeinflussen könnte, dass ein valider Wert ausgegeben wird.

5.1.1. Messbare Komponenten

Linux-Kernel Die wichtigste Komponente im gesamten Software Stack ist der Linux-Kernel. Er kontrolliert den Zugriff auf sämtliche Hardware, inklusive dem Arbeitsspeicher. Durch diese zentrale Rolle ist es notwendig, eine genaue Aussage über die Identität des Kernels treffen zu können. Ein Angreifer, welcher Kontrolle über den Kernel erlangt, hat freie Handhabe über alle Komponenten des Systems. Der Angreifer ist in der Lage, den Speicher von allen aktiven Prozessen auszulesen und zu manipulieren. Weiterhin kann er dies auch auf Dateien auf der Festplatte anwenden. Durch die durchgeführten Messungen ist allerdings kein Schutz vor Schwachstellen der Implementierung gegeben. Wenn ein Angreifer durch die Ausnutzung eines Fehlers in dieser Kontrolle über den Kernel erlangt, lässt sich dies nicht verhindern, da der Version des Kernels vertraut wird. Das führt dazu, dass Kernel versionen, die häufig genutzte Schwachstellen besitzen beispielsweise bei der Remote Attestation als nicht vertrauenswürdig deklariert werden sollten.

Dieses Problem besteht allerdings nicht nur in mobilen Geräten. Es wird ein Kernel eingesetzt, der ebenso in konventionellen Systemen genutzt wird und als gesamtes Image binär gemessen werden kann. Daher kann für die Messung das etablierte Konzept von PCs durchgeführt werden.

Bibliotheken Die Libraries bestimmen das Verhalten des Systems und sollten daher gemessen werden. Sie werden von beliebigen Prozessen genutzt und stellen so ein beliebtes Ziel für Angriffe dar. Kann ein Angreifer beispielsweise die SSL-Bibliothek austauschen, so ist es ihm möglich, sämtliche Aufrufe der Funktionen der Bibliothek zu manipulieren. Besonders im Fall der SSL-Bibliothek, welche die Verschlüsselung der Daten steuert, ist dies ein großes Problem. Sind die dort definierten Mechanismen manipuliert, ist es dem Angreifer möglich, Einsicht in Daten oder Übertragungen des Benutzers zu erlangen, während dieser keine Möglichkeit hat, dies zu erkennen.

Allerdings könnte die große Anzahl der Bibliotheken auf einem mobilen Gerät zu Problemen führen. Sollte vor jedem Aufruf einer Bibliothek eine Messung durchgeführt werden, kann der Overhead zu signifikanten Verzögerungen im Programmablauf führen. Dies wird während der Tests in Abschnitt 7 evaluiert.

Dalvik VM Die Dalvik VM muss als spezielle Bibliothek in jedem Fall gemessen werden, da sie den Start der Android-Anwendungen kontrolliert. Nauman et. al [16] beschreiben allerdings, dass eine einfache Messung der Virtuellen Maschine nicht ausreicht und dass eine Messung aller von der Maschine geladenen Java-Klassen notwendig sei, um das Verhalten der VM identifizieren zu können. Sie beschreiben weiterhin eine Möglichkeit, Dalvik mit einem entsprechenden Verfahren auszustatten. Dieses wird durch eine Erweiterung der Dalvik VM erreicht, welche in den Ladeprozess aller Klassen eingreift und diese misst. Dies umfasst so nicht nur die Anwendungsklassen, welche über die Packages installiert werden, sondern auch die auf dem mobilen Gerät genutzten Standard-Java-Klassen. Durch dieses Vorgehen wird allerdings ein sehr hoher Aufwand betrieben, da sämtliche Klassen gemessen und als Eintrag des SML gespeichert werden müssen. Dies kann auch durch das Messen der VM sowie der ausgeführten Android-Packages abgedeckt werden, daher wird die Messung der Klassen nicht durchgeführt.

Dalvik-Executables Wie beschrieben sollten die Dalvik-Executables gemessen werden. Diese befinden sich zwar, wie in Abschnitt 3.1.2 beschrieben in einem abgegrenzten Bereich. Dennoch ist eine manipulierte Anwendung dort eine Gefahr für die Sicherheit der Daten des Benutzers.

Zusammenfassend werden die einzelnen Komponenten tabellarisch aufgezeigt (Siehe Tabelle 5.1).

5.1.2. Durchführung von Messungen

Nachdem nun die RTM sowie die zu messenden Komponeneten festgelegt worden sind, muss noch geklärt werden, wie die einzelnen Messungen durchgeführt werden können. Dies ist wiederum abhängig von der messenden Komponente. Für die Messungen aus dem Kernel gibt es zwei mögliche Wege. Zum einen können aus dem Init-Prozess heraus Messungen durchgeführt werden. Dazu muss dieser so angepasst werden, dass vor der

Komponente	Bewertung der Messung	Informationen,
Kernel	essentiell	Misst weitere Komponenten, das Vertrauen dieser Messungen ist vom Kernel abhängig
Libraries	wichtig	Binäre Messung möglich, eventuell nur sicherheitsrelevante Bibliotheken messen, um Geschwindigkeit zu bewahren
Dalvik	notwendig	Weitere notwendige Komponente, da Messung von Anwendungen durchgeführt werden
Anwendungen	wichtig	von Anwendung abhängig
Klassen	unrentabel	hoher Aufwand, wenig Verbesserung

Tabelle 5.1.: Zusammenfassung der messbaren Komponenten

Ausführung jedes Prozesses eine Messung durchgeführt werden muss. Dies erfordert einen hohen manuellen Aufwand, da sämtliche Messungen manuell spezifiziert werden müssen. Dadurch ist dieses Vorgehen sehr fehleranfällig, da leicht Prozesse und genutzte Libraries übersehen werden können. Außerdem ist der Wartungsaufwand sehr hoch, da bei jedem hinzugefügten Programm und jeder Bibliothek auf der Kernelebene ein Messungseintrag hinzugefügt werden muss. Auch Änderungen und Löschungen ziehen manuellen Aufwand nach sich.

Als Alternative kann der Kernel modifiziert werden. Bei diesem Ansatz misst der Kernel Dateien, sobald diese ausgeführt werden. Diese Lösung nutzt den Zugriff des Kernels auf sämtliche Ressourcen des Systems. Da er immer involviert ist, wenn ein Prozess gestartet oder eine Datei gelesen werden muss, ist es möglich, in diesem Zuge eine Messung durchzuführen. Dies hat den Vorteil, dass durch aktivieren eines modifizierten Kernels alle Programme und Bibliotheken gemessen werden können, ohne dass der Betreiber des Gerätes feine Einstellungen vornehmen muss. Die Wartbarkeit des Systems im Gegensatz zu dem Init-Prozess ist deutlich höher, da keine manuelle Interaktion bei den Messungen notwendig ist. Nachteilig ist jedoch, dass aufgrund der fehlenden Einstellungsmöglichkeiten eventuell zu viele Messungen durchgeführt werden könnten, welche das System verlangsamen ohne die Sicherheit zu erhöhen. Da dies auch mit der Anzahl der Bibliotheken zusammenhängt, muss in den Tests überprüft werden, wie stark diese Messungen das System verlangsamen.

Für die Messungen der Bibliotheken, der Programme und der Dalvik VM wird der Kernel genutzt. Dieser wird so modifiziert, dass er die Messungen automatisch durchführt sobald ein Programm ausgeführt oder auf eine Bibliothek zugegriffen wird.

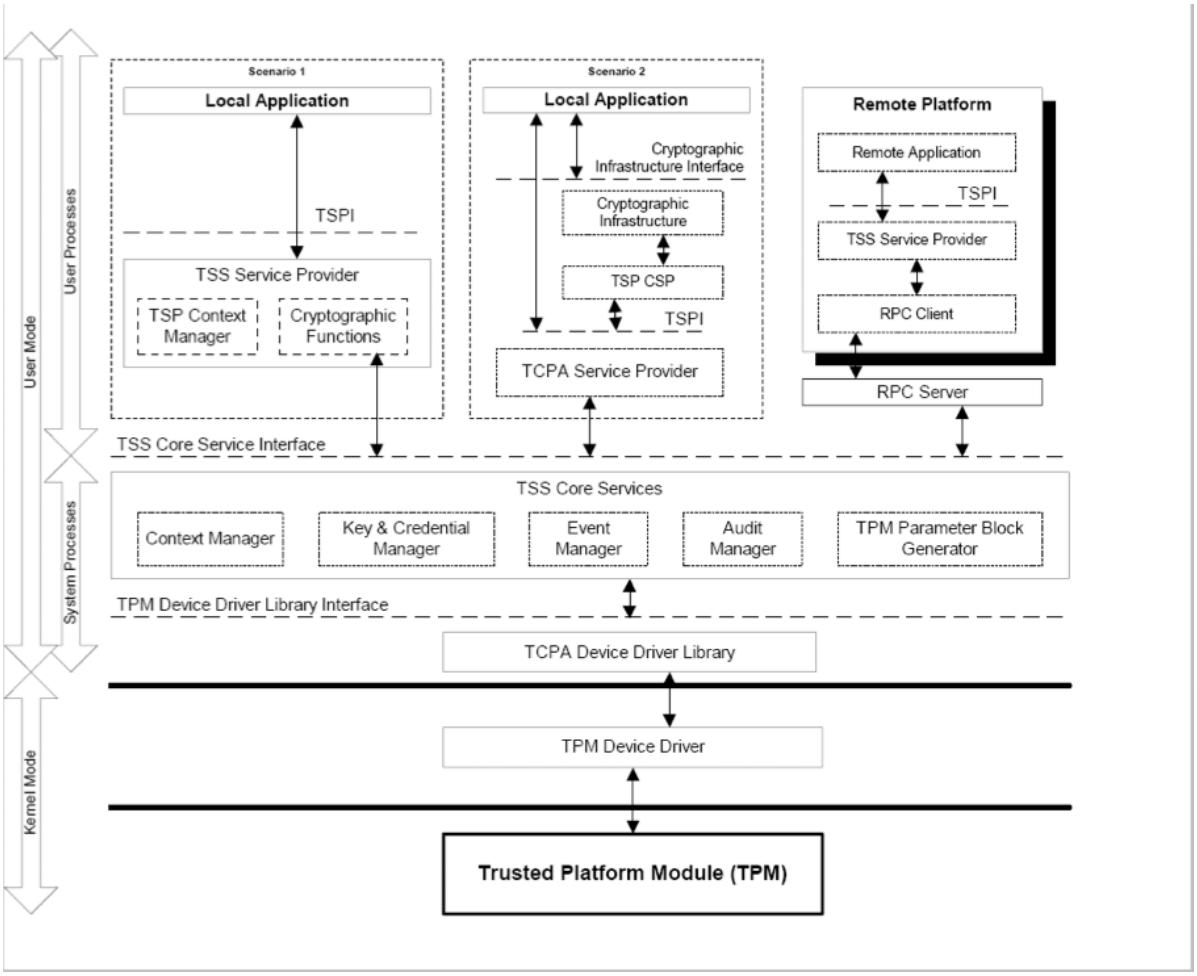

Abbildung 5.2.: TCG Software Stack [22]

5.2. TCG Software Stack

Als weitere Komponente wird ein TCG Software Stack (TSS) benötigt, der eine Schnittstelle zwischen Anwendungen und dem TPM darstellt. Der TSS muss auf allen Ebenen des Android Software Stack verfügbar sein.

Die TCG [22] spezifiziert einen Stack, welcher für den Zugriff auf das TPM genutzt werden kann. Dieser Stack wird in Abbildung 5.2 dargestellt. Er besteht aus mehreren Schichten, welche jeweils eigenen Aufgaben erfüllen:

Der TCG Service Provider (TSP) ist dafür zuständig, dass die Anwendung, die auf das TPM und seine Funktionen zugreifen will, eine Schnittstelle nutzen kann. Diese abstrahiert von den technischen Hintergründen, wie zum Beispiel der Verbindung mit dem TPM. Das Design ist so angelegt, dass ein TSP sich auch auf einem von dem TPM entfernten Rechner befinden kann.

So liesse sich die Dalvik VM auch als entfernter Rechner betrachten, da aus der VM ein direkt Zugriff auf Ressourcen des Systems nur eingeschränkt möglich ist. Da aus der Dalvik VM allerdings Netzwerkverbindungen erlaubt werden, ist ein solcher Zugriff möglich.

Der TSP greift auf die Dienste der TCG Core Services (TCS) zu. Während der TSP für den Zugriff von Anwendungen Schnittstellen liefert und von der Verbindung abstrahiert, erlauben die TCS den Zugriff auf das TPM und abstrahieren von dem Zugriff auf das Gerät. Die Spezifikation [22] beschreibt dies wie folgt:

> The TCG Core Services (TCS) provides a common set of services per platform for all service providers. Since the TPM is not required to be multithreaded, it provides threaded access to the TPM.

Ein „service provider“ ist nach der Spezifikation jede Komponente, welche einen Zugriff auf die Funktionen des TPMs erlaubt, zum Beispiel ein TSP. Der TCS nutzt wiederum die TCG Device Driver Library (TDDL) um über den TPM Device Driver auf das TPM zuzugreifen.

Es muss also ein TSS auf der Linux-Schicht verfügbar gemacht werden, um allen dort gestarten Prozessen den Zugriff zu ermöglichen. Weiterhin muss auch eine Android-Java-Schnittstelle bereitstehen, so dass Anwendungen, welche von Benutzern installiert werden, die Informationen nutzen können.

5.3. Remote Attestation Prozess

Wie beschrieben, muss ein Prozess die Daten der Messungen an einen entfernten Verifier übertragen. Dies kann beispielsweise von einem TNC-Client durchgeführt werden.

Da sich eine parallel erstellte Bachelorarbeit mit der Integration von TNC auf Android-Geräten befasst, wird dies nicht weiter betrachtet. Es werden allerdings die weiteren, für die Remote Attestation notwendigen Komponenten, wie die Chain of Trust und der TSS für diesen Client bereitgestellt.

Abbildung 5.3 zeigt die Abgrenzung der dafür notwendigen Komponenten. In Orange sind die in der vorliegenden Arbeit erzeugten Komponenten markiert, während die grünen Komponenten in der Bachelorarbeit betrachtet werden. Für den TSS wird im Client eine Android-Schnittstelle erzeugt, welche eine Remote Attestation durch eine Java-Anwendung ermöglicht. Aufgrund ihres Umfangs wird diese auf die nötigen Funktionen beschränkt. Für einen möglichen Client auf der Linux-Ebene soll ebenfalls eine Schnittstelle bereitgestellt werden.

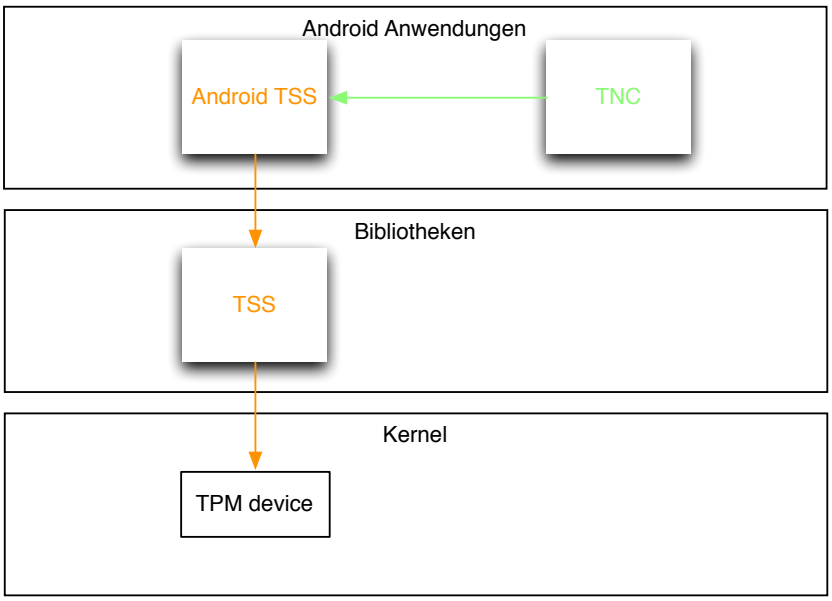

Abbildung 5.3.: Abgrenzung der Komponenten für die Remote Attestation

5.4. Installationsprozess

Der Installationsprozess muss so angepasst werden, dass er Messungen durchführt und diese in der Metrik speichert. Die Metrik-Schutzmechanismen müssen so eingestellt sein, dass sie dem Installationsprozess einen schreibenden Zugriff erlauben.

Der Prozess schreibt Informationen, wie den Hashwert der APK und den Speicherort und Namen des installierten Packages. In der folgenden Tabelle ist ein Beispiel für die Daten der Metrik dargestellt.

File	/data/app/de.fhhannover.inform.trust.tsse.apk
Hash-Wert	2b3eb073a3f62e03d9460be63b015d3a40d60f78

Falls der Installationsprozess ein Update einer bestehenden Anwendung durchführen soll, muss dieser auch die Metrik dementsprechend anpassen. Es ist also nötig, die Metrik in einem solchen Fall zu lesen, den Wert zu löschen und durch den aktuelleren zu ersetzen. Bei der Deinstallation einer Anwendung muss der Messwert aus der Metrik entfernt werden.

5.5. Prozess zur Durchsetzung der sich aus der Metrik ergebenden Regeln

Der Verifier ist als Prozess dafür zuständig, Programme zu messen und vor ihrer Ausführung mit den in der Metrik gespeicherten Werten zu vergleichen. Dazu muss er als Flaschenhals allein für die Ausführung von Programmen verantwortlich sein. Nur so ist es möglich,

Anwendungen, bei manipulierten Werten, von der Ausführung abzuhalten.

Ein Start darf nur möglich sein, wenn der Programmname und Speicherort in der Metrik vorhanden und passend sind. Weiterhin muss der dazu gemessene Hashwert übereinstimmen. Eine Messung der Anwendung durch die Komponenten der Chain of Trust und eine darauf folgende Abfrage des PCR-Registers und des Measurement Logs ist möglich. Dabei würde durch den Verifier die Messung angestoßen und durch die entsprechende Messkomponente durchgeführt. Anschließend muss der Verifier den Wert von der Messkomponente abfragen, und ihn dann mit der Metrik vergleichen. Nach Berechnung des Hashergebnisses ließe sich dann entscheiden, ob die Anwendung gestartet werden soll.

Eine andere Möglichkeit ist, dass die Messungen von dem Verifier selbst durchgeführt werden, um unabhängig von anderen Komponenten zu bleiben und die Struktur des Systems durch Einhaltung der Schichtentrennung übersichtlich zu lassen. Wie Abbildung 5.4 zeigt, führt eine Messung durch die Chain of Trust Komponenten zu einem komplizierterem Aufbau, der deutlich mehr Komponenten involviert. Daher wird das direkte Messverfahren eingesetzt, obwohl es eigene Messungen benötigt.

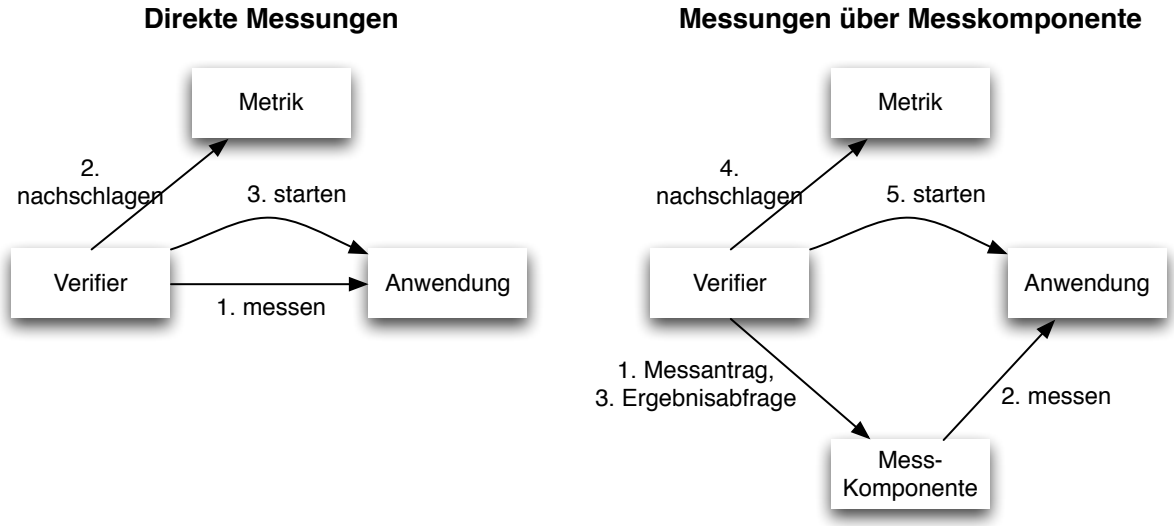

Abbildung 5.4.: Mögliche Verifikationsabläufe

Für die direkte Verifikation existieren zwei Alternativen für den Verifier: die Dalvik VM und der Kernel.

Der Kernel kann die Durchsetzung der Metrik direkt durchführen. Nachdem eine Anwendung gemessen wurde, kann dieser die Metrik laden, den Wert für die Anwendung auslesen und die Werte vergleichen. Der Kernel ist anschließend in der Lage die betroffenen Prozesse eigenhändig zu beeinflussen. So kann er einen gestarteten Anwendungsprozess bei einer erkannten Manipulation einfach abbrechen.

Andererseits kann auch die Dalvik VM genutzt werden, um die Anwendungen, welche gestartet werden sollen, zu messen und mit der Metrik abzugleichen. Dazu ist sie besonders geeignet, da sie durch die Eigenschaften der Architektur allein für die Ausführung der Java-Anwendungen verantwortlich ist.

Dies ermöglicht auch eine einfache Mitteilung eines möglichen Abbruchs an den Benutzer. Sollte der Kernel einen solchen Abbruch durchführen, ist eine Mitteilung an den Benutzer nicht möglich. Der Nutzer sollte allerdings darüber aufgeklärt werden, aus welchem Grund seine Anwendung nicht ausgeführt werden konnte. Dies ist über die von Dalvik bereitgestellten Mechanismen, beispielsweise dem Werfen einer entsprechenden Exception, möglich.

Diesen beiden Lösungsansätzen, bei denen die mit der Durchsetzung der Regeln beauftragte Komponente die Messungen eigenständig durchführt, können prinzipiell als gleichwertig angesehen werden. Die Aufgabe dieser Komponente ist bei beiden Ansätzen klar umrissen. Da es die Mitteilung an den Benutzer ermöglicht wird für den Prototypen diese Messung und Durchsetzung aus der Dalvik VM getätigt.

Eine Möglichkeit, die Sicherheit zu erhöhen, ist die Integration der in der MTM Spezifikation [25] genannten RTM+RTV. Diese setzen sehr nahe an der Hardware an und starten nur verifizierte Komponenten. So kann sichergestellt werden, dass die Dalvik VM als Verifier und der Installationsprozess in einer bestimmten Version vorliegen. Dies liesse sich auch durch Secure und Authenticated Boot erreichen. Es ist aber möglich, den hier beschriebenen Verifier als Rim_Auth in der MTM Spezifikation vorgesehenen Kette einzuordnen und so das System abzurunden. Nötig wäre hierzu nur eine von dem Gerätehersteller erzeuge Root of Trust.

5.6. Metrik

Die lokale Verifizierung wird mit Hilfe einer Metrik durchgeführt, in der alle installierten Programme aufgeführt sind. Relevant ist vor allem der Speicherort der Metrik. Wie beschrieben kann innerhalb des Android-Software-Stack sowohl die von Android bereitgestellte Datenbank als auch eine Datei im Dateisystem genutzt werden.

Als weitere mögliche Ablage der Messwerte bietet sich das TPM an. Solange das System nicht neu gestartet wurde, ist die Kontrolle der Messwerte beim Start der Anwendung mit Hilfe des SMLs möglich. Da bei einem Neustart alle TPM-Informationen gelöscht werden, gehen die Installationsinformationen verloren. Das führt dazu, das diese Metrik mit erlaubten Messwerten außerhalb des TPMs abgelegt werden muss.

Dies führt zu der Entscheidung, dass die Metrik nicht im TPM gespeichert werden kann. Wichtig für die Funktion der lokalen Verifizierung ist, dass die Messwerte, mit denen verglichen wird, integer bleiben. Daher müssen diese innerhalb der Metrik gegen Manipulationen geschützt sein. Dies kann mit Linux Mechanismen durchgeführt und mit Trusted Computing Mechanismen verstärkt werden.

Wenn die Metrik beispielsweise im lokalen Dateisystem abgelegt wird, ist eine Zugriffsbeschränkung durch die Linux-Standardmechanismen wichtig. Dazu ließe sich eine Gruppe anlegen, welche schreibenden und lesenden Zugriff auf die Metrik erhält. Der Installationsprozess und der Verifier müssten dann dieser Gruppe angehören. Dieses Verfahren ist nicht ausreichend, da alle Prozesse mit root-Rechten auf die Metrik zugreifen können. Wenn es einem Angreifer gelingt, durch einen Exploit in einem Prozess an root-Rechte zu gelangen, ist der gegebene Schutz wirkungslos.

Eine Alternative zu dieser Methode ist Mandatory Access Control (MAC). Bei einer MAC-Lösung lässt sich genau spezifizieren welche Prozesse Zugriff auf eine Datei bekommen sollen. Es ist darüber hinaus auch möglich die Art des Interagierens mit der Datei feingranular anzupassen. Damit wäre sichergestellt, dass nur durch den Installationsprozess und den Verifier die Metrik geschrieben oder gelesen wird. Wenn jedoch eine dieser zwei Komponenten kompromittiert wurde, ist auch MAC wirkungslos. Um einen solchen Fall zu verhindern, kann aber auf die erweiterten Sicherheitsmechanismen von Trusted Computing zurückgegriffen werden.

Dazu bietet sich zum Beispiel Sealed Storage, was von Löhr zusätzlich zu Secure Boot als Pattern definiert wird, an. Sealed Storage kann genutzt werden, um die Metrik auf der Festplatte verschlüsselt abzulegen. Eine wichtige Eigenschaft von Sealed Storage ist die Verschlüsselung anhand einer gewünschten Zielkonfiguration. Nur wenn diese gegeben ist, wird die Metrik entschlüsselt. Als mögliche Zielkonfiguration könnte ein PCR mit dem Messwerten des Installationsprozesses und des Verifiers gefüllt werden. Nur wenn diese beiden Komponenten unverändert sind, kann die Metrik entschlüsselt werden.

Für eine gegebene Zielkonfiguration wird ein Wert angegeben, welchem der PCR-Wert zur Entschlüsselungszeit entsprechen muss. Da wie in Abschnitt 2.3 die PCR-Werte von der Ausführungsreihenfolge abhängen, führt dies zu einem Problem. Es gibt eine Arbeit, die sich mit diesem Problem beschäftigt [29]. Sie betrachtet, wie das Sealing und Unsealing auf Systemen mit wechselnden PCR-Werten gehandhabt werden kann. Dies ist auf mobilen Plattformen interessant, da sich die Ausführungsreihenfolge der Anwendungen durch den Benutzer häufig ändert. Als alleinige Lösung reicht Sealed Storage allerdings nicht aus, da während der Laufzeit des Betriebssystems die Metrik unverschlüsselt vorliegen muss. Eine Kombination von MAC und Sealed Storage würde zu einer sicheren Lösung des Problems führen. Weiterhin muss, wie beschrieben, sowohl der Installationsprozess als auch der Verifier vor manipulation geschützt sein.

5.7. Technisches Konzept

Das technische Konzept beschreibt, wie die im Konzept beschriebenen Schritte auf ein Android Mobiltelefon angepasst werden können. Es wird besonders darauf eingegangen, welche Technologien eingesetzt werden können, um die Integration zu erreichen. Weiterhin werden diese Technologien mit Alternativen abgewogen und deren Einsatz begründet. Abbildung 5.5 zeigt die vereinfachte Struktur der Komponenten ohne Modifikationen. Auf der untersten Schicht befindet sich der Kernel, mit dem TPM-Device. Dieses wird für das Konzept angenommen und für den Prototypen durch einen Emulator bereitgestellt. Die darüber liegende Schicht bildet die Dalvik VM. Der Hauptprozess der Vm, der Zygote, startet die auf der obersten Schicht angesiedelten Anwendungen.

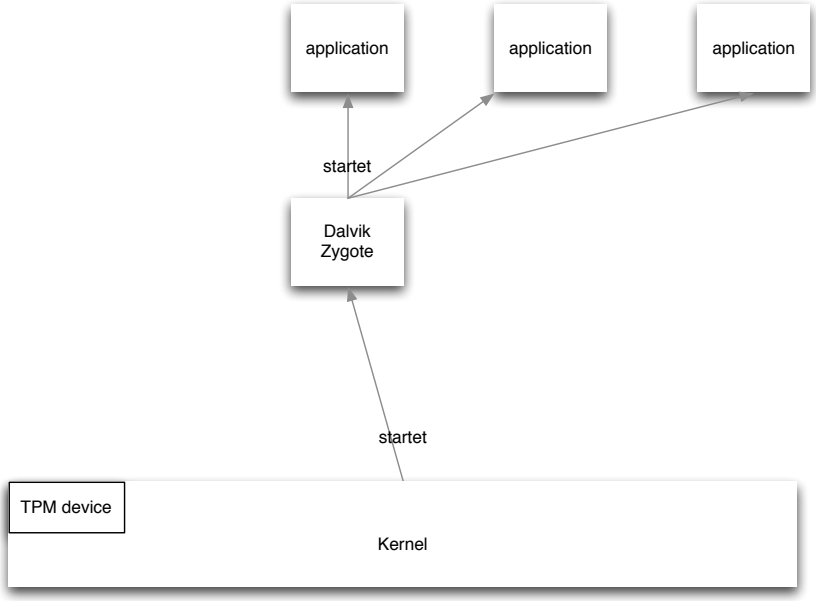

Abbildung 5.5.: Basisstruktur der Android Architektur

Secure Boot und Authenticated Boot

Trotz des Ausschlusses von Secure und Authenticated Boot für die Implementierung eines Prototypen, sind Lösungen bekannt, welche genutzt werden können, um diese Mechanismen zu implementieren. Anstatt einer hardwarenahen Root of Trust for Measurement, muss aber dennoch im weiteren Verlauf der Arbeit der Linux Kernel als RTM angenommen werden. Mit entsprechender Hardware und Herstellerunterstützung ist eine Abbildung eines geschützten Bootprozesses möglich.

ARM bietet mit ihrer Trust Zone[1] [3] eine Möglichkeit, um Secure Boot sicher auf ARM Architekturen zu integrieren. Diese Technologie sorgt dafür, dass auf Hardwareebene zwei „Welten“ geschaffen werden: eine normale und eine sichere Welt. Dazu werden Arbeitsspeicher, Bussystem und Prozessor so angepasst, dass zwischen den Welten unterschieden werden kann. So ist es möglich den Zugriff auf Ressourcen der sicheren Welt einzuschränken und zu kontrollieren.

Johannes Winter [28] beschreibt die Möglichkeit, Secure Boot durch eine Verknüpfung der MTM-Technologien und der ARM Trust Zones abzubilden. In der vorgestellten Lösung wird zuerst ein Betriebssystem in der sicheren Welt gestartet, welches als Hypervisor für die normale Welt genutzt wird und zusätzlich Software-MTMs enthält. In der normalen Welt wird nach einer Messung durch die sichere Welt ein Gast-Betriebssystem gestartet, welches den TCG Software Stack bereitstellt und über eine Schnittstelle mit dem sicheren Betriebssystem kommunizieren kann.

[1] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/CACGCHFE.html

Chain of Trust

Die Messungen durch die in der Arbeit angenommenen RTM kann wie beschrieben durch eine Kernelerweiterung erfolgen. IBM hat mit der Integrity Measurement Architecture (IMA)[2] eine solche Kernelerweiterung implementiert [19]. IMA ist ein Security-Kernelmodul, welches sich in die Systemaufrufe einhängt und Messungen von ausführbaren Dateien durchführt. Weiterhin misst IMA alle geladenen Kernelmodule und Libraries. Neben den automatisch durchgeführten Messungen ist auch die Definition von manuellen Messungen möglich. Dazu wird ein virtuelles Dateisystem erstellt, in welches ein *measure request* abgelegt werden kann. Ein measure request besteht aus einem File-Descriptor und einem Tag. Der von der Anwendung geöffnete File-Descriptor gibt die Datei an, welche gemessen werden soll. Das Tag beschreibt die Messung. Sobald ein measure request in das Dateisystem gelegt wird, wird durch IMA eine Messung der Datei durchgeführt und diese in das TPM geschrieben.

Die Dalvik VM ist das auf den Kernel folgenden Kettenglied der Chain of Trust. Sie nutzt díe IMA measure requests, um Messungen von Anwendungen abzustoßen. So werde alle in dem System gestarteten Anwendungen und Bibliotheken gemessen. Die um IMA erweiterte Struktur wird in Abbildung 5.6 dargestellt.

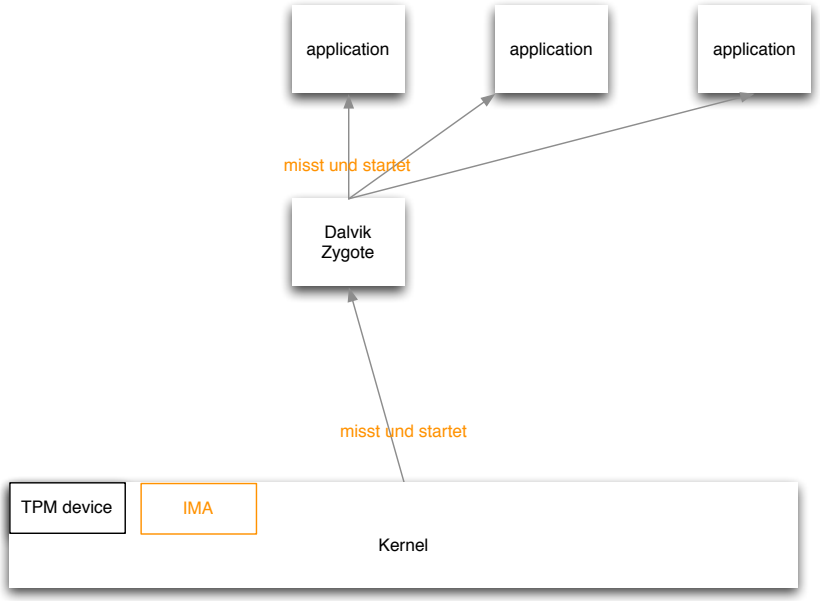

Abbildung 5.6.: Basisstruktur mit Messkomponenten

[2] http://linux-ima.sourceforge.net/

TCG Software Stack

Für den TSS gibt es bereits eine Reihe von Lösungen, welche auch auf dem Linux Betriebssystem eingesetzt werden können. Eine bekannte Open-Source-Implementierung ist TrouSerS[3]. TrouSerS bietet einen Prozess für die Kommunikation mit dem TPM und eine C-Bibliothek, mit der Highlevel Programme auf diesen Prozess zugreifen können.

Um den TSS auch für Android-Anwendungen verfügbar zu machen, müssen Java-TSS-Lösungen betrachtet werden, die sich in die Android-Architektur einbetten lassen. Zwei dafür geeignete Implementierungen sind *tpm4java*[4] und *TPM/J*[5], zwei reine Java-Entwicklungen, welche auf das TPM-Interface */dev/tpm* zugreifen und die Kommunikation, ähnlich wie TrouSerS, direkt ausführen. Diese Programme sind beide Open-Source. Als Nachteil ist das Alter der Projekte zu sehen, da die letzte Version von *tpm4java* im Jahre 2006 und die von *TPM/J* im Jahre 2007 erschienen ist. Das größere Problem stellt jedoch der direkte Zugriff auf das TPM-Device dar. Der direkte Zugriff auf diese Ressourcen wird von dem Android-Framework nicht unterstützt. Daher müsste für eine Nutzung aus einer Android-Anwendung eine Implementierung des internen Device-Treibers erstellt werden.

Als weitere Alternative existiert der von dem Institute for Applied Information Processing and Communications[6] entwickelte jTSS[7]. Wie tpm4java gibt es eine jTSS-Version, welche die Kommunikation mit dem TPM eigenständig regelt. Diese Version leidet unter dem gleichen Nachteil, dass für die Android-Plattform eine eigenständige Implementierung eines Device Treibers nötig ist. Allerdings bietet das IAIK auch einen jTSS-Wrapper an, welcher über eine mit JNI angesprochene C-Bibliothek auf den TrouSerS TSS zugreift um mit dem TPM zu kommunizieren. Auch wenn primär an dem jTSS entwickelt wird, ist dieses Projekt weiterhin aktuell. Die letzte Änderung am jTSS ist im Jahre 2010 durchgeführt worden. Der jTSS-Wrapper ist dual lizensiert und für Open-Source-Entwicklungen kann die GNU GPL v2 angewandt werden.

Zusätzlich besteht eine Möglichkeit, auf einen von TrouSerS erstellten TCP Socket zuzugreifen, auf welchem die Funktionen bereitgestellt werden. Mit dem Aktivieren einer Android-Permission, welche den Netzwerkzugriff erlaubt, ließe sich aus Android auf diesen Socket zugreifen und so das Protokoll implementieren. Nachteilig hieran ist, dass zusätzlich zu der vollständigen Android-Schnittstelle das Netzwerkprotokoll implementiert werden muss. Da mit dem jTSS-Wrapper eine Lösung zur Verfügung steht, die den wenigsten Implementierungsaufwand nach sich zieht, wird diese für die Integration ausgewählt.

Einen weiteren Grund für diese Entscheidung stellt die Aktualität des Projekts dar, da die Software getestet und gewartet wird. Weiterhin ist so der Aufbau der TPM-Integration darauf fokussiert, bestehende Open-Source-Lösungen zu kombinieren und zu integrieren.

[3] http://trousers.sourceforge.net/
[4] http://tpm4java.datenzone.de/trac
[5] http://projects.csail.mit.edu/tc/tpmj/
[6] http://www.iaik.tugraz.at/
[7] http://trustedjava.sourceforge.net/

Abbildung 5.7 zeigt wie sich der Trusted Software Stack in die Android Architektur einfügt.

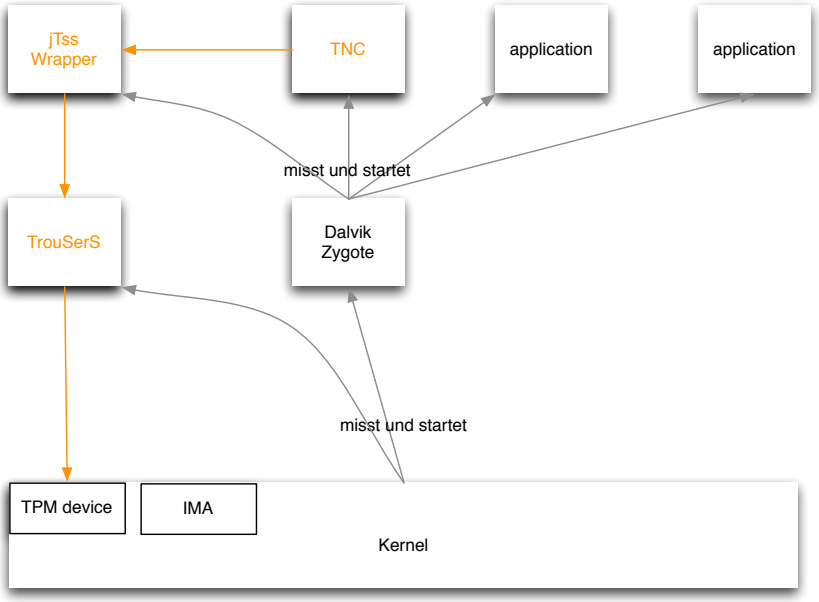

Abbildung 5.7.: Komponenten inklusive Trusted Software Stack

Installationsprozess

In Android wird zur Installation der *installd* Prozess verwendet. Dieser kann für die Benutzung der lokalen Verifizierung angepasst werden. Abbildung 5.8 zeigt den als Linux Prozess gestarteten Installationsdienst.

Prozess zur Durchsetzung der sich aus der Metrik ergebenden Regeln

Wie in Abbildung 5.9 zu sehen, ist der durch den Android Software Stack vorgegebene Flaschenhals die Dalvik VM. Die Dalvik VM ist für die Ausführung aller Java basierten Anwendungen zuständig. Daher muss der Dalvik Hauptprozess, der Zygote genannt wird, so angepasst werden, dass er vor der Ausführung eine Messung durchführt und diese mit der Metrik vergleicht. Durch diese spezielle Einschränkung, ist eine lokale Verifizierung möglich.

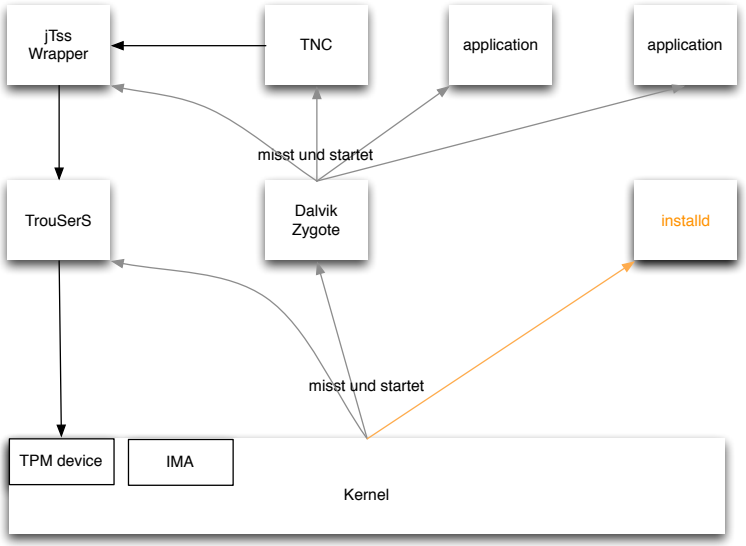

Abbildung 5.8.: Komponenten mit Installationprozess

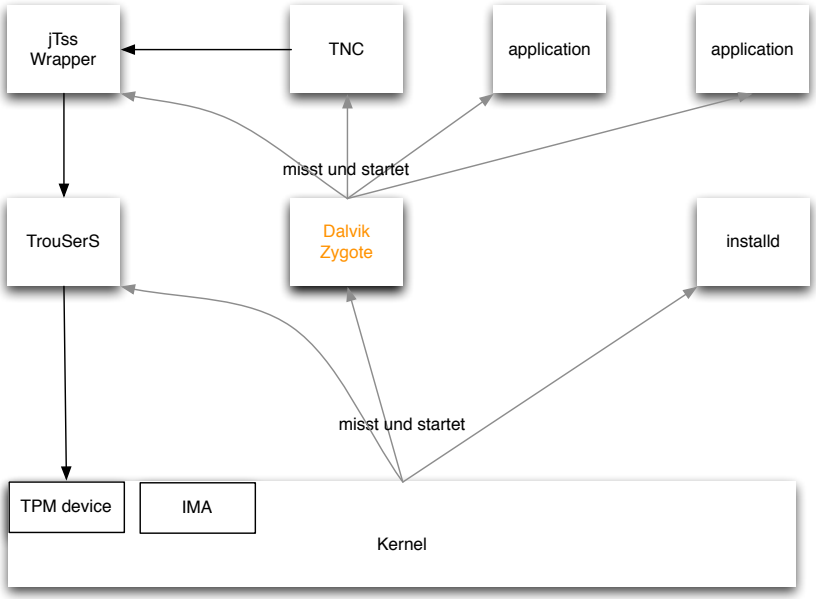

Abbildung 5.9.: Komponenten mit modifiziertem Verifier

Metrik

Für die Metrik wird eine einfache binäre Datei verwendet, die wie beschrieben gegen Angriffe geschützt werden kann. Als Alternative wäre die Nutzung der android-eigenen Datenbank möglich, was aber die Abschottung der Metrik gegenüber unbefugten Zugriffen erschweren würde. Weiterhin ist der Schutz der Metrik dann von der Sicherheit der Datenbank abhängig. Um den Schutz der Dateisystem-Metrik zu gewährleisten, ist nur die Sicherheit der vier unmittelbar betroffenen Komponenten nötig: der Metrik-Datei, des Installationprozesses, des Verifiers und des Kernels. Da eine Datenbank ein komplexes Softwaresystem darstellt, ist dieses schwer zu sichern und kann auch unerkannte Fehler enthalten, welche eine Kompromittierung erleichtern. Dadurch wird auch der Schutz des Systems erschwert. Aus diesem Grund wird für die Metrik statt dieser Alternative eine Datei genutzt. Abbildung 5.10 stellt abschließend alle Komponenten, mitsamt der benutzen Metrik dar.

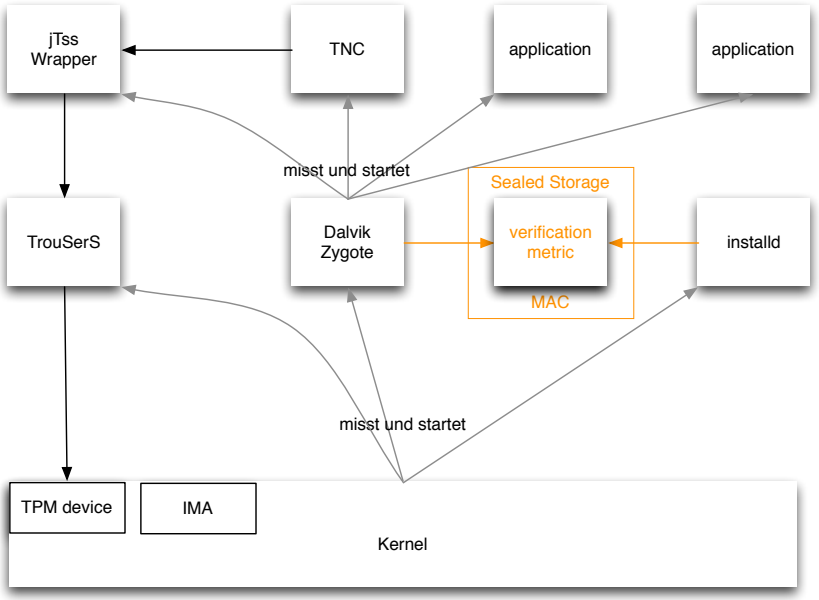

Abbildung 5.10.: Komplette Struktur

5.8. Zusammenfassung des Konzeptes

Das während der Masterarbeit erstellte Konzept zur Integration der Trusted Computing Technologien beschreibt Komponenten, welche die in Kapitel 4 dargestellten Lösungsansätze bereitstellen.

Das Konzept zeigt weiterhin auf, wie diese Komponenten auf einem Android-Mobiltelefon in die bestehende Architektur aufgenommen werden können. Wie in dem Abschnitt 1.2 beschrieben, existiert ein Ansatz, welcher sich ebenfalls mit der Integration von Remote Attestation Mechanismen in die Android-Plattform befasst [16]. Dabei wird ein spezialisierter TCG Software Stack und ein angepasster TPM-Emulator definiert, welche auf der obersten Ebene des Android Software Stacks angeordnet sind. Diese beiden Komponenten werden seperat von der Kernelmessung betrieben. In der vorliegenden Arbeit wird dies auf andere Art gehandhabt. Es wird sowohl für die Messungen der vom Kernel gestarteten Prozesse als auch der Anwendungen der Dalvik VM der modifizierte IMA-Kernel verwendet. Daher werden alle Messungen für die Remote Attestation durch eine einzelne Komponente durchgeführt und können so zentral gespeichert und eingesehen werden. Weiterhin fällt der Aufwand für die Messungen jeder Klasse weg, was die Geschwindigkeit des Systems verbessern sollte.

Bei allen Komponenten die zu der Android Architektur hinzugefügt werden, handelt es sich um bekannte Open-Source-Implementierungen, welche auf Spezifikationen der Trusted Computing Group beruhen. Weiterhin beschreibt diese Arbeit einen Mechanismus zur lokalen Verifizierung sowie für Secure Boot, was in der Lösung von Naumann et al. nicht betrachtet wird.

6. Implementierung des Prototypen

Dieser Abschnitt beschreibt die Implementierung und Portierung der im Konzept erläuterten Komponenten. Dabei werden nach und nach die in Abbildung 5.10 dargestellten Teile in ein Android System integriert.

6.1. Secure Boot

Die für Secure Boot nötigen Implentierungsschritte setzten unterhalb der vorhandenen Komponenten an. Daher wurde keine Implementierung unternommen.

Es wurden jedoch die in den Android-Quellen existierenden Bootloader untersucht. Weitere Schritte, um diese anzupassen, finden aus den in Kapitel 5 genannten Gründen nicht statt.

Chain of Trust

Der Kernel, der die RTM abbilden soll, muss modifiziert werden, um die in Abschnitt 5.7 beschriebenen Funktionen zu erhalten. Dazu werden die Android Kernelsourcen[1] verwendet und so gepatched, dass sie das IMA-Kernelmodul enthalten. Der so kompilierte Kernel führt selbstständig Messungen durch, welche in das TPM geschrieben werden. Er speichert seine Daten in einem virtuellen Dateisystem ab, welches durch Modifikation des *init.rc*-Skripts nach `/sys/kernel/security/ima` gemounted wird.

Die Funktionen der ADB-Konsole sind besonders sinnvoll, wenn die Ausgaben der Hintergrundprozesse und Module überprüft werden sollen. Dies gilt natürlich auch für die Messungen von IMA. Die Datei `ascii_runtime_measurements` in dem gemounteten Verzeichnis kann genutzt werden um die IMA-Messungen anzuzeigen. Tabelle 6.1 zeigt einen Auszug der ersten zwanzig Messungen.

Die Reihenfolge der Messungen ist dabei gut zu erkennen. Der erste Prozess, welcher gestartet wird, ist *init*. Dieser startet wiederum weitere Prozesse, welche Libraries benötigen. Hinter dem Namen */system/bin/app_process* verbirgt sich der Dalvik VM-Hauptprozess. Dieser ist wiederum für die Messung und den Start der Java-Anwendungen zuständig.

Die Dalvik VM musste für Messungen der Android-Pakete angepasst werden. Um eine solche Messung durch IMA zu erlauben muss ein *measure_request* in das virtuelle Dateisystem geschrieben werden. Dazu wurde die VM so verändert, dass ein solcher Request erstellt wird, wenn ein Paket von der VM geladen wird.

[1] http://android.git.kernel.org/?p=kernel/linux-2.6.git

PCR	Messwert	Datei
10	1a783edb5103a201b180f920b033c26e48d9e179	/init
10	c28916493bffe9333883026f28a1f9701aae1359	/system/bin/servicemanager
10	588962d72b08735155dce379687a63403c74b50b	/system/bin/debuggerd
10	2972fab34a09a7cfd9e32608752f76ad05cd5ece	/system/bin/linker
10	d0c3eabf3f928c60c3918f8c0b943d81f9384a50	/system/bin/rild
10	3360aa9904264a9f4530b1cda80e55311b0e958d	/system/bin/sh
10	9f86aa15100d0f24a51c4af4d052cbc43cbc0f43	/system/bin/app_process
10	dc4ebfb93f6e24e35b2c603b3aa01be027ce3fd5	/system/bin/mediaserver
10	7db93bff2e3ac63010e61df76773c80f9ca97ef3	/system/bin/vold
10	dc0a5c0f031a93b55321504fb5f951ccbe02b698	/system/lib/liblog.so
10	fca732b892c6000c259c8e1a281653f29fcd784d	/system/bin/installd
10	0928414af9ade3bde73bcdaec7bb30a35d5eb6cf	/system/bin/keystore
10	15f821e21fb141d6a4f56e06c2327241268612ff	/system/bin/dbus-daemon
10	e2f43be1e7c983e451f2d6699cd3b2c47528f0f9	/system/bin/qemud
10	2517b3e7fcf567f3bd9a15a7457b9a0035b7120f	/system/lib/libc.so
10	030588456392caf07b9e8ad7c59596c9dd35199b	/system/bin/logcat
10	74ea20e2a5ffab928e3b95a8193fa4defa30b838	/system/lib/libstdc++.so
10	678ed59919cade831e6341a5b0c222bfc9714cbc	/system/lib/libm.so
10	8e94002a5c483778561c81fb75b8805d24a8af74	/system/lib/libexpat.so
10	10d298d28e6861df8a651be6760935f6b1a3a794	/system/lib/libaudioflinger.so
10	c8e44ffce4dca63132bb33a545bbfde28edc5b4b	/sbin/adbd

Tabelle 6.1.: Auszug der IMA-Messungen

Die hierfür benutzte Methode befindet sich in der Klasse *dalvik.system.DexFile*. Diese Klasse besitzt einen JNI-Teil, welcher modifiziert wurde, da aus der C-Umgebung ein Zugriff auf das Dateisystem einfacher durchzuführen war.

In der Datei *android/dalvik/vm/native/dalvik_system_DexFile* wurde die Funktion `Dalvik_dalvik_system_DexFile_openDexFile` modifiziert. Es wurde eine Funktion `int measure (const char* filename, const char* tags)` erstellt und aufgerufen um Messungen an IMA zu senden. Das Ergebnis dieser Modifikation ist in einem weiteren Auszug aus den IMA-Messungen (Tabelle 6.2) zu erkennen.

Nach den Bibliotheken werden die grundlegenden Android-Pakete gestartet. Die Benutzerapplikationen werden zum Systemstart nicht mitgeladen und sind daher nur dann zu finden, wenn sie manuell aktiviert wurden. Bei einem normalen Systemstart finden inklusive aller grundlegenen Android-Anwendungen etwa einhundert Messungen statt.

Diese Messungen müssen nur dann wiederholt werden, wenn sich eine Datei geändert hat. Andernfalls führt IMA zu einem *measure request* keine Messung doppelt aus. Daher ist die größte Einschränkung des Systems beim Systemstart zu erwarten. Im laufenden Betrieb werden deutlich weniger Anwendungen kurz nacheinander gestartet.

PCR	Messwert	Datei
10	1a9d1ffe9ce0a6e9bac8498e00d1b3c521ed8e41	/system/lib/egl/libGLES_android.so
10	3a18f86ac4290623dbe98b0d83d8cca93b2319f7	/system/bin/bootanimation
10	a22d040081102bdb8c29108d4cc0daddbf8bdf31	/system/app/SettingsProvider.apk
10	c34b93776e6aa015be5ff2b252524861d4adf1f5	/system/lib/hw/sensors.goldfish.so
10	bc879164bd47e7b8671e4b8536e97116b24b9681	/system/lib/libsoundpool.so
10	a947650a86b405ae19ff7a7d4aff8e73076b3085	/system/app/LatinIME.apk
10	77c7f073d027a22eb1e35c5e95ce1c42858b574a	/system/lib/libjni_latinime.so
10	19ef56b1b4422974b3f5e0553fe76832820b78a7	/system/app/Launcher.apk
10	2f38f051e83b839fe366fd33d2200ec0d9fd1f78	/system/app/Phone.apk
10	e58b24ab8c8557762b809497eb3274b54be27464	/system/app/GlobalSearch.apk
10	f4d61852e86bc0814966877b421c2e0db5746224	/system/app/TelephonyProvider.apk

Tabelle 6.2.: Auszug der IMA-Messungen mit Android-Paketen

Durch die vorhandene Akkulaufzeit ergibt es sich, dass eine mobile Plattform selten gebootet wird. Daher ist der Aufwand einer Messung zum Bootzeitpunkt selten und kann unter Umständen vernachlässigt werden.

6.2. Trusted Platform Module

Da das Trusted Platform Module momentan nicht als Hardwarekomponente zur Verfügung steht, muss es durch eine Anwendung emuliert werden. Für diesen Zweck bestehen schon einige Anwendungen, von denen eine auf die Android-Plattform portiert wurde. Der von der ETH Zürich entwickelte Emulator[2] besteht aus drei Komponenten: einer Anwendung, die auf einem Linux-Socket die TPM Funktionalität anbietet, einer TPM Device Driver Library sowie einem Kernel Modul, welches das Device `/dev/tpm` anlegt.

Dieses Kernel-Device wird von dem Trusted Software Stack verwendet, um auf den Emulator zuzugreifen. Weiterhin muss auch IMA über dieses Device die Verbindung mit dem TPM aufbauen. Diese Interaktionen werden in Abbildung 6.1 gezeigt.

Die Verbindung zwischen dem TPM Emulator und dem IMA-Kernel unterliegt einigen Schwierigkeiten. Als Kernelmodul wird IMA gestartet, bevor der Init-Prozess weitere Prozesse starten kann. Daher ist IMA bereits verfügbar, wenn der Emulator noch gestartet wird. Da dieses Verhalten in einem normalen System erwünscht ist, um alle Prozesse vor dem Start messen zu können, soll dies auch beibehalten werden. Allerdings führt es dazu, dass am Anfang des Bootvorgangs Messungen nicht in dem Emulator gespeichert werden können. Dieses Problem ist aber von dem Emulator abhängig, so dass es nicht betrachtet werden muss, wenn ein Hardware TPM vorhanden ist.

[2] http://tpm-emulator.berlios.de/index.html

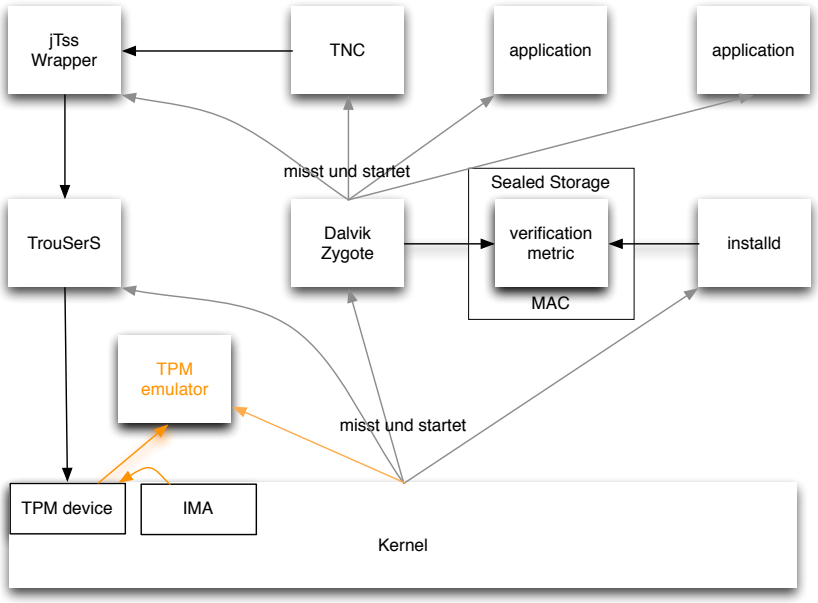

Abbildung 6.1.: Struktur mit TPM Emulator

6.2.1. Verbindung von IMA und dem TPM

Die Verbindung zwischen dem TPM-Emulator und der IMA-Kernelerweiterung lässt sich auf mehrere Arten durchsetzen. Eine dieser Möglichkeiten wäre, den von dem TPM-Emulator erstellten Socket in IMA zu nutzen und direkt anzusprechen. So ist es möglich, die Existenz des Emulators festzustellen, sobald eine Verbindung auf den Socket aufgebaut werden kann. Nachteilig bei dieser Interaktion ist, dass das Protokoll dieser Verbindung auch in IMA implementiert werden muss.

Der Linux Kernel bietet die Möglichkeit einer Definition von TPM Treibern. IMA greift standardmäßig auf diese zu. Die Schnittstelle des TCG_TPM-Moduls beinhaltet zwei Funktionen:

```
int tpm_pcr_read(u32 chip_id, int pcr_idx, u8 * res_buf, int
    res_buf_size);
int tpm_pcr_extend(u32 chip_id, int pcr_idx, const u8 * hash);
```

IMA benutzt `tpm_pcr_read` um festzustellen, ob ein TPM verfügbar ist. Im Normalfall erzeugt IMA einen Ausnahmefehler, wenn diese Prüfung fehlschlägt und IMA sich nicht im Testmodus befindet. Um dieses Problem zu lösen, muss die Integrity Measurement Architecture so erweitert werden, dass sie die Messungen vorhält, bis ein TPM verfügbar ist. Das Modul bietet seinerseits eine Schnittstelle an, die von Treibern genutzt werden kann, um Geräte hinzuzufügen. Ein solcher Treiber kann geschrieben werden, um den Emulator als Gerät zu registrieren. Für diesen Treiber lässt sich das ohnehin schon bestehende Device des TPM-Emulators nutzen. Der vom TPM-Emulator erstellte Kerneltreiber muss so modifiziert werden, dass es über die Schnittstelle des TCG_TPM-Moduls das bestehende Device */dev/tpm* registriert. Damit ist keine weiteren Modifikationen an IMA nötig.

Die Möglichkeit des Zugriffs über den Socket des TPM-Prozesses fällt auf Grund des Aufwandes und der gut durchführbaren und in das Design passenden Alternative aus. Daher wird der TPM-Emulator wie beschrieben über den Kerneltreiber modifiziert.

6.2.2. Zurückhalten von Messwerten

Das Zurückhalten der Messwerte kann entweder in IMA selbst, oder wenn vorhanden in dem TPM-Emulator Treiber durchgeführt werden. Da IMA eigenständig eine Liste aller Messungen verwaltet, ist es einfach, diese so zu verändern, dass die *extend*-Operationen erst dann durchgeführt werden, wenn ein TPM vorhanden ist.

Ein Treiber ist von der Zuständigkeit her besser geeignet, die Messungen vorzuhalten. Für IMA sollte die Art des Trusted Plattform Module unerheblich sein und es sollte auch keine Modifikationen enthalten, die auf spezielle TPM-Eigenschaften reagieren. Dennoch werden in diesem Fall alle Messungen doppelt gespeichert, was einen erhöhten Speicherbedarf nach sich zieht. Da gerade auf mobilen Geräten der RAM eine kritische Größe einnimmt und häufig kein Auslagerungsspeicher zur Verfügung steht, wird diese Liste nur im IMA-Kernel gespeichert.

IMA Header

Die IMA bietet eine Header-Datei, welche Funktionen für die IMA definiert, mit denen auf das TPM zugegriffen werden kann. Dabei wird das „TCG_TPM“-Modul gekapselt, um das Lesen und Erweitern des TPMs zu ermöglichen, weiterhin wird eine Funktion bereitgestellt, um anzuzeigen, ob ein TPM vorhanden ist.

Listing 6.1: IMA-Header Funktionen

```
static inline int have_tpm(void )
{
#if defined (CONFIG_TCG_TPM) || defined (CONFIG_IMA_EMULATOR_MODE)
 int err;
 if ((err = tpm_pcr_read(IMA_TPM, 0, NULL, 0)) == 0)
 {
   return 1;
 } else {
   //ima_info("ERROR: %d\n", err);
 }
#endif
 return 0;
}
```

Die `have_tpm()`-Funktion (Siehe Listing 6.1) nutzt die im TCG-Modul definierte Funktion `tpm_pcr_read()`. Wenn diese keinen Fehlercode ausgibt, wird davon ausgegangen, dass das TPM verfügbar und funktionstüchtig ist. Sollte ein Fehler ausgegeben werden oder kein „TCG_TPM“-Modul aktiviert sein, wird die Funktion eine 0 zurückgegeben.

IMA-Queue

Die in der IMA verwendete Queue enthält wie beschrieben alle Messungen der Architektur. Das Standard-Verhalten ist ein direktes Aufrufen der *extend*-Operation auf dem TPM.

Sie wird so angepasst, dass sie erst nach dem Aktivieren des TPM hinzugefügt werden. Dazu wird die Queue so umgebaut, dass die Daten, welche in der Funktion `ima_add_measure_entry` hinzugefügt werden, nur gesichert werden und erst dann ein *extend* durchgeführt wird, wenn das TPM verfügbar ist. Listing 6.2 stellt die Modifikationen an der Queue dar.

Listing 6.2: IMA-Queue modifikationen

```
/**
* In emulator mode, it is necessary to wait until a tpm is ready.
*/
#ifdef CONFIG_IMA_EMULATOR_MODE
  if(!ima_emulator_ready)
  {
    /**
    * if a tpm is available, the measurements are inserted.
    */
    if(have_tpm())
    {
      ima_used_chip = 1;
      ima_info("IMA␣extending␣list␣to␣TPM␣emulator␣\n");

      list_for_each(pos,&ima_measurements)
      {
        tmp= list_entry(pos, struct ima_queue_entry, later);

        ima_extend(tmp->entry->digest);
      }

      ima_emulator_ready = 1;
      goto out;
    }
    else
    {
      goto out;
    }
  }
#endif
ima_extend(entry->digest);
```

Die in der Kernelkonfiguration definierte Option `CONFIG_IMA_EMULATOR_MODE` wird genutzt, da das Vorhalten der Messwerte nur nötig ist, wenn der Kernel mit einem Emulator betrieben wird. Um einen erhöhten Aufwand der Abfragen zu verhindern, sobald ein Emulator verfügbar ist, wird die `ima_emulator_ready`-Variable verwendet. Ist diese ungleich 0, werden die darauf folgenden Abfragen übersprungen und das IMA-Normalverhalten wird genutzt.

Um festzustellen, ob ein TPM verfügbar ist, wird die in der *ima.h* definierte Funktion `have_tpm()` verwendet. Diese gibt 1 zurück, falls ein TPM vorhanden ist. In diesem Fall wird für die Elemente der Liste, der Einfügereihenfolge entsprechend, `ima_extend` aufgerufen. Gibt `have_tpm()` 0 zurück, wird keine Aktion unternommen und die Funktion wird beendet.

6.3. Basisdienste für Remote Attestation

Dieser Abschnitt beschreibt, wie die für die Remote Attestation nötigen Komponenten vorbereitet und in die Android-Architektur eingebunden wurden.

6.3.1. TCG Software Stack

Um die vollständige Funktionalität abzubilden und sie auch für beliebige Anwendungen verfügbar zu machen, wird ein TCG Software Stack (TSS) benötigt. Ein TSS kapselt alle Funktionen des TPM und stellt sie durch eine API für Programme zur Verfügung. Als freie Implementierung eines TSS wurde TrouSerS verwendet und für die Android Plattform portiert.

TrouSerS besteht aus mehreren Komponenten, welche die Spezifikation des TSS abbildet (Siehe 5.2). Die Hauptkomponente ist ein Dienst, welcher die TCG Core Service-Funktionen anbietet. Dieser wird TCSD genannt. Der TCSD nutzt die von TrouSerS implementierten TCS-Bibliothek und die TCG Device Driver Library (TDDL) für die Verbindung zum TPM und definiert seine Funktion anhand einer TCG Service Provider (TSP) Schnittstelle.

So können beliebige Anwendungen auf der Linux-Ebene der Plattform die TPM-Funktionen nutzen. Diese Funktionen sollen auch für Anwendungen auf der Android Applikationsebene verfügbar sein. Die TSP-Schnittstelle wird also im Android Framework nutzbar gemacht. Um dies vorzubereiten, müssen zwei Schritte durchgeführt werden. Zuerst ist es nötig aus der Java VM auf die TrouSerS TSS-Schnittstelle zuzugreifen und sie in der VM abzubilden. Anschließend werden Android-Service-Definitionen erstellt, damit beliebige Anwendungen darauf zugreifen können.

Wie in Abschnitt 5.7 beschrieben, stellt der jTSS Wrapper eine API bereit, die den TrouSerS Dienst über JNI in Java verfügbar macht. Der Wrapper wird über das Android Native Development Kit für die Plattform kompiliert, und kann anschließend aus der Anwendung genutzt werden.

Um von dem jTSS-Wrapper auf TrouSerS zugreifen zu können, muss innerhalb des Android-Manifests eine Permission aktiviert werden. Da die beiden Prozesse über einen Socket kommunizieren, ist es nötig, dass die Internet-Permission vorhanden ist.

```
<uses-permission android:name="android.permission.INTERNET"/>
```

6.3.2. Definition der Schnittstellen

Die geschichtete Architektur von Android und die darin eingebettete Lösung für das Trusted Computing führen zu einer Vielzahl von Schnittstellen, die sowohl von externen Prozessen als auch innerhalb der Lösung genutzt werden. Diese sollen hier beschrieben werden.

Android Schnittstelle

Die innerhalb von Java angebotene Schnittstelle für die Remote Attestation ist in ihrem Funktionsumfang recht eingeschränkt gegenüber den vom TPM angebotenen Mechanismen. Dennoch werden alle benötigten Funktionen angeboten: es ist notwendig, einen PCR-Wert an den Verifier zu übertragen. Um vor Manipulationen und Replay-Attacken geschützt zu sein, wird ein Quote verwendet.

Listing 6.3: Quote.java

```
public class Quote implements Parcelable {
        public int pcrValueSize;
        public byte[] pcrValue;
        public int signatureSize;
        public byte[] signature;
        public int replayProtectionSize;
        public byte[] replayProtection;
}
```

Dieser Quote besteht, wie in Listing 6.3 gezeigt, aus einem Zufallswert, welcher die Eindeutigkeit jedes Quotes sicherstellen soll, um Replay-Angriffe abzuwehren. Der Verifier muss Quotes aussortieren, bei denen dieser Wert bereits vorgekommen ist, da sie von einem Angreifer stammen können, welcher ein Paket mit einem gültigen Wert erneut versendet, um den Status einer nicht gültigen Plattform zu verschleiern.

Weiterhin enthält der Quote den PCR-Wert, welcher vom Verifier anhand des SML und der Policy überprüft werden muss.

Zusätzlich werden die Daten signiert und die Signatur wird als Teil des Quotes versendet. Dafür wird ein Schlüssel benötigt. Bei diesem Schlüssel kann es sich entweder um einen, durch eine entsprechende CA, signierten Attestation Identity Key oder um einen durch Direct Anonymous Attestation erstellten Schlüssel handeln.

Die im Quote verwendeten Größenangaben werden gebraucht, um die Daten als Android-Parcel zu versenden. Listing 6.4 zeigt, wie die Daten in ein Paket geschrieben werden.

Listing 6.4: Paket schreiben

```
public void writeToParcel(Parcel out, int arg1) {
        if(pcrValueSize == 0){
                pcrValueSize = pcrValue.length;
        }
        out.writeInt(pcrValueSize);
        out.writeByteArray(pcrValue);
        if(signatureSize == 0){
                signatureSize = signature.length;
```

```
            }
            out.writeInt(signatureSize);
            out.writeByteArray(signature);
            if(replayProtectionSize == 0){
                    replayProtectionSize = replayProtection.length;
            }
            out.writeInt(replayProtectionSize);
            out.writeByteArray(replayProtection);
    }
```

Listing 6.5 zeigt die Erstellung eines Quotes von einem Parcel.

Listing 6.5: Aus Paket lesen

```
    public void readFromParcel(Parcel in) {
            pcrValueSize = in.readInt();
            pcrValue = new byte[pcrValueSize];
            in.readByteArray(pcrValue);
            signatureSize = in.readInt();
            signature = new byte[signatureSize];
            in.readByteArray(signature);
            replayProtectionSize = in.readInt();
            replayProtection = new byte[replayProtectionSize];
            in.readByteArray(replayProtection);
    }
```

Um Fehler mit Speicherzugriffen auszuschließen, wird die Größe genutzt um die Array-Größen vor dem Beschreiben zu definieren.

Für die Verfahren der Attestation muss jeweils eine Interaktion mit vertrauenswürdigen Instanzen durchgeführt wird. Der AIK muss beispielsweise von dem TPM erzeugt werden, um dann an eine AIK CA geschickt zu werden. Alle Informationen, die für die CA nötig sind, müssen zu einem Paket zusammengestellt werden. Dieses besteht mindestens aus dem öffentlichen Schlüssel des AIK sowie dem öffentlichen Teil des Endorsement Keys.

Alternativ muss die Erzeugung des DAA-Geheimnisses ermöglicht und für die Attestation bereitgestellt werden. Die Implementierung der Schnittstellen für beide Verfahren wird im Rahmen der Arbeit nicht durchgeführt. Stattdessen wird für jedes Quote ein Schlüssel genutzt, der vorher in das TPM geladen wurde. Mit diesem wird der PCR-Wert anschließend signiert. Dieses Verfahren bietet keinen Schutz der Privatsphäre, dennoch zeigt es die Machbarkeit der Integration.

Um an den für die Erstellung der Schlüssel notwendigen SRK zu gelangen muss dieser von TrouSerS geladen werden. Die Schlüssel werden von TrouSerS in der Datei `System.data` abgelegt. Der Speicherort dieser Datei muss in der TrouSerS-Konfiguration angepasst werden, da sie normalerweise in der Systempartition abgelegt ist. Da diese allerdings nur read-only gemounted wird und die Datei auch beschrieben werden muss, muss sie in das schreibbare Verzeichnis `/data` verlegt werden.

Nach dieser Änderung lässt sich der SRK laden, um den Schlüssel zu erzeugen. Die Erzeugung scheiterte allerdings an einem Authentisierungsfehler des Schlüssels. Da dieser Fehler aus zeitlichen Gründen nicht vollständig nachvollzogen werden konnte, kann

keine Signatur erstellt werden. Dieses Problem wird im Zuge der Implementierung der Attestierungsverfahren behoben werden müssen, da auch dort eine Schlüsselerzeugung notwendig ist. Die vollständige Implementierung aller TSS Methoden stellt keine Herausforderungen dar und kann nachgelagert durchgeführt werden.

Die weitere implementierte Methode der Schnittstelle ist für die Beschaffung des SML zuständig. Das SML wird durch die `getSml`-Methode erzeugt. Diese erzeugt ein *EventLog*, welches alle PCR-Messwerte und weitere Informationen enthält. Jede Messung wird als *Event* in einem Array in dem EventLog gespeichert. Dies wird in Listing 6.6 dargestellt.

Listing 6.6: Attribute des Events

```
public class Event implements Parcelable{

        public long pcrIndex;
        public long eventType;
        public int valueSize;
        public byte[] value;
        public int eventDataSize;
        public byte[] eventData;
}
```

Das Attribut `pcrIndex` beschreibt, in welches PCR die Messung eingefügt wurde. Der `eventType` beschreibt den Typ des Events. Das Attribut `value` enthält den Messwert des Events, und `eventData` weitere Daten.

IMA erzeugt das *EventLog*, welches über TrouSerS ausgegeben wird. Dazu musste auf der mobilen Plattform die Konfiguration des TCSD angepasst werden, um das Log aus der Datei `/sys/kernel/security/ima/binary_runtime_measurements` auszulesen. Da allerdings keine BIOS-Boot-Messungen zur Verfügung stehen, erzeugt der Versuch des Auslesens einen Fehler.

Durch die Beschränkung des Lesevorgangs auf das von IMA befüllte PCR 10 kann dieser Fehler vorübergehend umgangen werden. Leider wird trotz korrekter Konfiguration kein *EventLog* ausgelesen. Dieses Problem konnte während der zur Verfügung stehenden Zeit nicht behoben werden. In diesem Fall gibt die Schnittstelle ein leeres *EventLog* zurück.

Abbildung 6.2 zeigt die Ausgabe der Schnittstellen-Testapplikation.

Weitere Schnittstellen

Die Schnittstellen des jTSS-Wrappers sowie von TrouSerS enthalten die von der TCG spezifizierten Funktionen. Dabei bietet der jTSS-Wrapper im Gegensatz zu TrouSerS ein objektorientiertes Interface. Um dieses zu nutzen, muss ein Context erzeugt und verbunden werden, wie in Listing 6.7 dargestellt. Anschließend wird das TPM-Objekt von dem Context bereitgestellt. So kann der PCR-Wert ausgelesen werden.

Listing 6.7: Nutzung des jTSS-Interfaces 1

```
context = contextFactory.newContextObject();
context.connect();
TcITpm tpm = context.getTpmObject();
```

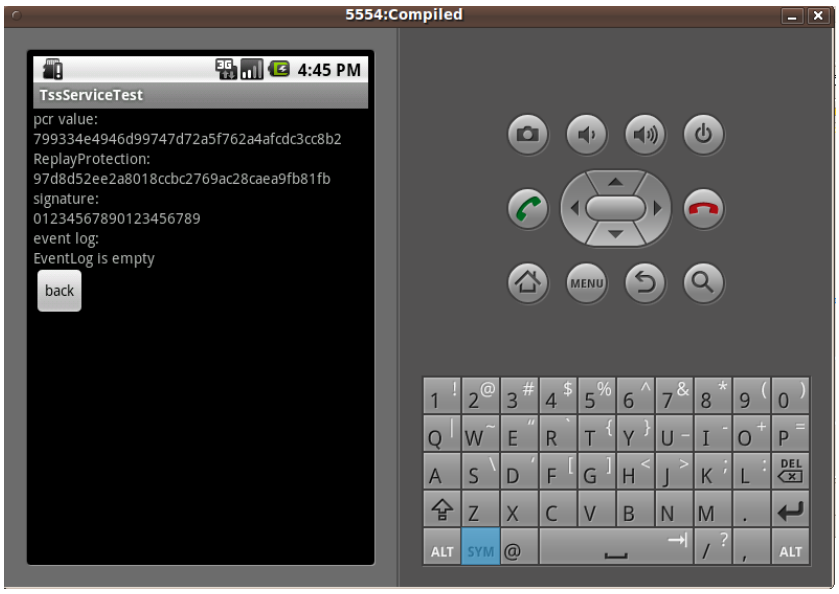

Abbildung 6.2.: Schnittstellen Testapplikation

```
TcBlobData pcrV = tpm.pcrRead(10);

Log.i("getPcrQuote", String.format("PCR␣10␣has␣value␣%s", pcrV.
    toHexString()));
```

Dabei wird über den Context das TPM-Objekt zurückgegeben, auf welchem dann die `pcrRead`-Methode aufgerufen werden kann.

TrouSerS bietet die gleichen Methoden an, verhält sich jedoch nicht objektorientiert. TrouSerS hält sich mit den definierten Funktionen nah an der TCG-Spezifikation [22]. Zum Vergleich von TrouSerS und jTSS wird in Listing 6.8 die `Tcsip_PcrRead`-Funktion der TrouSerS Schnittstelle dargestellt.

Listing 6.8: TrouSerS PcrRead Funktiion

```
extern TSS_RESULT Tcsip_PcrRead
(
    TCS_CONTEXT_HANDLE    hContext,                    // in
    TPM_PCRINDEX          pcrNum,                      // in
    TPM_PCRVALUE*         outDigest                    // out
);
```

Diese entspricht der in der Spezifikation definierten Funktion `Tspi_TPM_PcrRead`, wohingegen im jTSS die Methode wie in Listing 6.9 definiert ist.

Listing 6.9: jTSS pcrRead Methode

```
public TcBlobData pcrRead(final long pcrIndex) throws TcTssException;
```

6.4. Local Verifier

Von den drei benötigten Teilen für die lokale Verifizierung ist die wichtigste Komponente die Metrik. Diese enthält die Zuordnung von Messwerten zu Anwendungen. Sie darf nur durch den für die Pflege vorgesehenen Installationsprozess geändert werden und sollte gegen unerlaubte Zugriffe und Änderungen geschützt werden.

Der Schutz der Datei ist von höchster Wichtigkeit, da eine kompromittierte Metrik die Funktion der Verifizierung völlig unbrauchbar machen würde. Die erwähnten Schutzmöglichkeiten sind Dateizugriffsrechte, Mandatory Access Control und Sealed Storage. Da der Installationsprozess als Prozess des `root`-Benutzers gestartet wird, wird nur für diesen das Schreib-Zugriffsrecht für die Metrik eingerichtet:

```
-rw-r--r-- root     root          3317 2010-07-28 08:30 metric
```

6.4.1. Library für Zugriff auf die Metrik

Die für die lokale Verifizierung notwendigen Funktionen werden durch eine einzelne statische Bibliothek bereitgestellt. Diese Bibliothek stellt drei Methoden zur Verfügung, welche in dem Quellcode-Abschnitt 6.10 aufgelistet sind.

Listing 6.10: metricd.h

```
/*
 * Metricd is used to create a metric, which contains all installed
    software on a mobile device.
 * This metric can be used to decide if a application is allowed to run
    .
 *
 * Author: Johannes Westhuis
 */
#ifndef __METRICD_DEF_
#define __METRICD_DEF_

int metricd_install(const char* filename, const char* futurename);
int metricd_remove(const char* filename);
int metricd_verify(const char* filename);

#endif
```

Diese drei Methoden haben folgende Aufgaben:

metricd_install(const char* filename, const char* futurename) Der übergebene Dateiname `filename` spezifiziert die Anwendung, welche gemessen werden soll. Diese wird von der Bibliothek gemessen und mit dem Dateinamen `futurename` zusammen in der Metrik gespeichert. Dieser Umweg wurde genutzt, da in der betreffenden Funktion des Installationsprozesses nur ein temporärer Name verwendet wird.

metricd_remove(const char* filename) Die remove-Funktion entfernt eine Messwert-Zuordnung aus der Metrik. Dazu wird der Dateiname in der Metrik gesucht und, falls er zu finden ist, entfernt. Wenn der Dateiname nicht vorhanden ist, wird keine Aktion durchgeführt.

metricd_verify(const char* filename) Die wichtigste Funktion prüft, ob ein Wert in der Metrik vorhanden ist. Zu diesem Zweck, wird die Metrik aus dem Dateisystem gelesen. Dabei wird eine Liste aller Messungen anhand der Metrik erzeugt. Die zu prüfende Datei wird innerhalb dieser Funktion gemessen. Anschließend wird in der Liste nach dem Dateinamen gesucht. Ist eine Zuordnung vorhanden, werden die Messwerte verglichen. Andernfalls wird ein Fehler-Code zurückgeliefert. Wird ein Unterschied beim Vergleich der Messwerte erkannt, erzeugt auch das die Rückgabe eines Fehler-Codes. Wenn alle Prüfungen erfolgreich durchlaufen wurden, gibt die Funktion den Wert 0 zurück. Durch die Nutzung verschiedener Rückgabewerte lassen sich genauere Fehlerangaben an den Benutzer der Plattform erstellen.

Abbildung 6.3 stellt die jeweiligen Abläufe der Bibliothek dar. Die *install*-Funktion lädt die angegebene Datei und misst sie. Anschließend wird die Metrik geladen und in einer Liste abgelegt. Die Liste wird gebraucht, da bei der Installation schon vorhandene Versionen überschrieben werden müssen und dieses mit einem Datencontainer leichter durchzuführen ist.

Wird im Verlauf der Installation in der Liste eine Version gefunden, wird diese entfernt. Danach wird der neue Messwert zu der Liste hinzugefügt. Zum Schluss wird die gesamte Liste in die Metrik-Datei zurückgeschrieben. Das Format der Datei ist an das IMA-Format angelehnt und setzt sich aus dem PCR, dem Messwert und dem vollständigen Dateinamen zusammen. Jede Zeile der Metrik stellt einen Messwert dar.

Die *remove*-Funktion hat einen ähnlichen Ablauf. Für sie ist es nicht nötig, eine Datei zu messen. Daher wird nur die Metrik geladen und die Dateizuordnung gesucht. Wenn eine Zuordnung gefunden wird, wird sie aus der Liste gelöscht. Abschließend wird die Metrik erneut zurückgeschrieben.

Für den *Verify*-Prozess ist ein etwas anderes Vorgehen notwendig. Hier wird zuerst die Metrik geladen und die Liste erzeugt, bevor die Datei gemessen wird. Das wird aus dem Grund getan, um die Zeit zwischen Messung der neuen Komponente und des Vergleichs mit einem Referenzwert möglichst klein zu halten. Dies soll eine „time of check to time of use“-Race Condition verhindern. Diese Race Condition wird von Bishop und Mangler [6] wie folgt definiert:

> [...] the time-of-check-to-time-of- use (TOCTTOU) flaws. A TOCTTOU flaw occurs when a program checks for a particular characteristic of an object, and then takes some action that assumes the characteristic still holds when in fact it does not.

TOCTTOU Schwachstellen entstehen also, wenn es einem Angreifer möglich ist, zwischen einem Test und dem Ausführen einer Aktion einzugreifen. Eine solche Race Condition kann zum Beispiel bei dem Zugriff auf eine Datei auftreten.

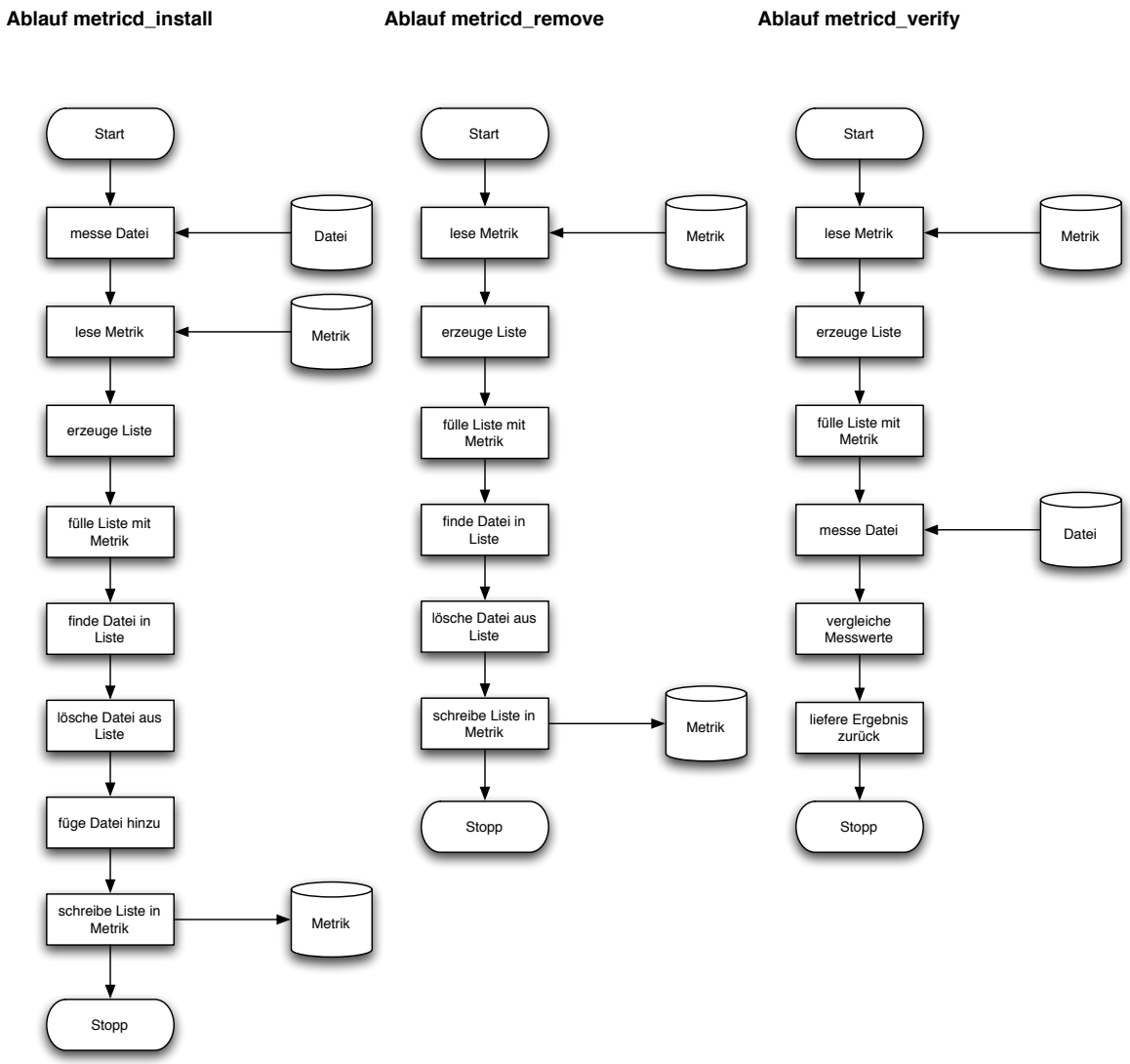

Abbildung 6.3.: metricd Abläufe

Wenn zwischen der Prüfung der Zugriffsberechtigung und dem Dateizugriff zu viel Zeit vergeht, könnten sich die Rechte an der Datei so geändert haben, dass der Zugriff nicht mehr erlaubt ist. Wenn sich die Anwendung ändert, nachdem sie vom Verifier gemessen wurde, würde ein Zugriff erlaubt werden, obwohl es sich um eine manipulierte Datei handelt. Durch die Verkürzung der Zeit zwischen Rückgabe des Vergleichs und damit der Entscheidung, was zu tun ist, und der Erstellung des Messwerts der Datei werden die Chancen, dass eine solche Schwachstelle ausgenutzt werden kann, verringert. Es ist so allerdings nicht möglich, sie vollständig auszuschließen, da zwischen der Entscheidung und dem Vergleich mehrere Rechenschritte liegen, in welchen die Datei kompromittiert werden könnten.

6.4.2. Erstellen einer RIM durch installd

Nach dem Erstellen der Bibliothek für die Metrik, stellt die Integration der Bibliothek in den Installationsprozess keine große Herausforderung mehr dar. Dafür ist ein Aufruf der Funktion `metricd_install` während der Installation sowie der Aufruf von der Funktion `metricd_remove` bei der Deinstallation nötig.

Um die geeigneten Stellen für diese Aufrufe festzustellen, musste der Installationsprozess *installd* untersucht werden. Er besteht aus zwei C-Dateien, `installd.c` und `commands.c`. In der Datei `installd.c` werden die in der `commands.c` implementierten, grundlegenden Funktionen zu abstrakteren Aufgaben zusammengefasst und ein Socket bereitgestellt, über welchen die Aufgaben angestoßen werden können. In der *commands.c* sind unter anderem Funktionen wie `is_valid_apk_path`, `install`, `uninstall` und `move_dex` definiert. `is_valid_apk_path` bekommt einen Pfad übergeben und überprüft diesen dahingehend, ob es sich um einen gültigen Pfad für ein Android-Package handelt. `install` erzeugt das Verzeichnis für ein Package auf der Plattform und legt die Benutzer- und Gruppen-Rechte für diesen Ordner an. `uninstall` löscht sämtliche Dateien und Verzeichnisse, die einem Package zugeordnet sind. `move_dex` entpackt die .dex-File und verschiebt sie in ein temporäres Verzeichnis. *installd* verwendet in der `install`-Funktion bis zum Abschluss der Installation temporäre Dateinamen, was dazu führt, dass der Metricd-Schnittstelle sowohl den temporären als auch den gewünschten Dateinamen übergeben werden muss. Die Deinstallation wird in der Funktion `uninstall` durch den Aufruf von `metricd_remove` durchgeführt.

Für eine Integration der MTM RIMs (Siehe 2.6.2) müsste anstatt der Metrik ein RIM sicher abgelegt werden, welches von dem Verifier vor dem Ausführen überprüft werden kann. Dies kann auch durch diese Metrik gewährleistet werden, da die Spezifikation keine genauen Implementierungen eines Verifiers vorgibt.

System Applikationen

Eine Reihe von Anwendungen wird durch die Android Quellen auf die Plattform kopiert. Diese durchlaufen nicht den Installationsprozess und werden daher auch nicht automatisch in die Metrik aufgenommen. Da diese aber dennoch von Dalvik geladen und gestartet werden, wird auch für sie eine Prüfung anhand der Metrik durchgeführt.

Unterbindet der Verifier nun die Ausführung einer solchen Anwendung, sind wichtige Systemfunktionen nicht mehr verfügbar. Eine solche Konstellation kann dazu führen, dass das System unbrauchbar wird. Um dies zu verhindern, müssen die Anwendungen in die Metrik aufgenommen werden.

Dazu wurde ein Programm geschrieben, welches die Bibliothek verwendet, um Dateien in die Metrik einfügen zu können. Dieses Programm muss nur einmal auf der Plattform ausgeführt werden, um die im Verzeichnis */system/app/* liegenden Android Packages zu messen und der Metrik hinzuzufügen.

Da die auf einem Standard-System verfügbaren Anwendungen nicht variieren, muss diese manuelle Installation nur einmal durchgeführt werden und kann als vorbereitete Metrik auf jede Plattform ausgeliefert werden. Die durch dieses Programm ausgeführten

Installationsvorgänge werden automatisch gemessen, was im Test in Kapitel 7.3 genutzt wird. Listing 6.11 zeigt, dass bei einer Messung eines Verzeichnisses zuerst die Datei in die Metrik installiert wird. Anschließend wird zur Überprüfung der Funktion eine Verifizierung durchgeführt.

Listing 6.11: Installation eines Verzeichnisses

```
char tmp [size];
// Erstellen des vollstaendigen Pfadnamens
snprintf(tmp, size, "%s/%s", filename, dir->d_name);

clock_t start = clock();

// Installation der Datei
metricd_install(tmp, tmp);
```

Das Installationswerkzeug ist vor allem für die Vorbereitung einer auslieferbaren Metrik, aber auch als Testwerkzeug konzipiert. Dies kann vereint werden, da das Werkzeug nicht auf eine laufende Plattform geladen werden soll. Es ist vielmehr dazu gedacht, eine Metrik für eine Standard-Konfiguration der System-Anwendungen bereitzustellen, welche unverändert ausgeliefert wird. Für die Tests eignet es sich besonders, weil viele Installationsdurchgänge von verschiedenen Dateien kurz aufeinander folgend durchgeführt werden.

6.4.3. Dalvik als RIM_Auth

Die Dalvik VM ist verantwortlich für die Verifizierung der Anwendungen. Bevor Dalvik eine Messung durch IMA erstellt, muss es anhand der Metrik überprüfen, ob sich die Anwendung in einem vertrauenswürdigen Zustand befindet. Dies wird durch den Aufruf der Bibliotheksfunktion `metricd_verify` durchgeführt. Da diese verschiedene Rückgabewerte besitzt, muss entsprechend auf diese reagiert werden.

Wird ein Fehlercode zurückgegeben, muss von Dalvik eine Exception geworfen werden, welche dem Benutzer angezeigt wird. Stimmen die Werte bei der Überprüfung überein, wird das Programm durch IMA gemessen und gestartet. Die von IMA durchgeführte zweite Messung dient der Vervollständigung der Informationen für die Remote Attestation und hat keinen Einfluss auf die lokale Verifizierung. Dies ist nötig, da die Remote Attestation ein vollständiges Bild der anfragenden Plattform benötigt, um eine Vertrauensentscheidung zu fällen.

Die Dalvik VM entspricht einem MTM RIM_AUTH. Nach der Überprüfung anhand eines RIMs müsste der Verifier das RIM signieren und mit `MTM_VerifyRIMCertAndExtend` in das MTM schreiben. Falls ein MTM genutzt werden soll, sind die nötigen Modifikationen an der Lösung sehr gering. Eine weitere IMA-Messung würde wegfallen, da die MTM-Funktion bereits ein Einfügen der Messwerte vorsieht.

6.5. Android Werkzeuge

Für das Deployment auf die Android Plattform sind verschiedene Werkzeuge nötig, welche von Android bereitgestellt werden. Diese unterstützen den Benutzer und Entwickler in vielen Bereichen: von der Implementierung über das Debugging bis hin zum Deployment auf dem Gerät.

6.5.1. Emulator

Mit das wichtigste Werkzeug, welches im Android Umfeld existiert, ist der Emulator. Er emuliert die benötigte Hardware und ermöglicht so einen realistischen Test des Android-Betriebssystems.

Für den Emulator werden so genannte Android Virtual Devices (AVD) gestartet. Ein AVD besteht aus mehreren Images: Dem Kernel, dem System, der Ramdisk, einem Benutzerdaten-Image sowie optional einem Image für eine SD-Karte. Erstellt man ein neues AVD, werden für diese jeweils Standard-Images benutzt. Für die vorliegende Arbeit war es allerdings erforderlich den Kernel, das System-Image sowie die Benutzerdaten anzupassen.

Bei dem System-Image handelt es sich um das Linux Basis-Dateisystem, welches während des Bootvorgangs schreibgeschützt gemounted wird. Es enthält die Bibliotheken und Binaries, welche zu dem Grundsystem gehören. Zum Beispiel den Init-Prozess und die Bionic-Libc sowie solche Bibliotheken und Binaries, welche von externen Quellen erstellt wurden, aber nur auf der Linux-Ebene genutzt werden.

Auf dem Benutzerdaten-Image befinden sich alle veränderlichen Daten, wie beispielsweise die Anwendungen des Benutzers sowie deren gespeicherte Daten. Dieses Image kann benutzt werden, um zusätzliche Programme, welche bei der Benutzung und Konfiguration des Systems helfen, zu speichern. Es wird für das Android-System als `/data` gemounted. Wie beschrieben wird in diesem Verzeichnis beispielsweise die Metrik abgelegt.

Das Kernel-Image enthält den Betriebssystem-Kernel, welcher im nächsten Abschnitt genauer betrachtet wird.

6.5.2. Android Software Development Kit

Das Android Software Development Kit ist als Plugin für die Entwicklungsumgebung Eclipse verfügbar. Es erweitert Eclipse um die Möglichkeit, Android-Projekte zu erstellen. Ein solches Projekt enthält die wichtigsten Projektkonfigurationen und eine vorgegebene, sinnvolle Ordnerstruktur. Weiterhin kompiliert Eclipse den Quellcode mit der auf Android angepassten Library und erzeugt Quellcode, beispielsweise für die Referenzierung von *Views*. Eine graphische Oberfläche zum Gestalten dieser Views wird zusätzlich bereitgestellt.

Außerdem kann durch das SDK eine AVD angegeben werden, auf welche die Anwendung geladen werden soll. So ist ein schnelles Testen der Anwendungen möglich.

6.5.3. Java Native Interface

Eine Besonderheit in dem Android Application-Framework besteht in der Möglichkeit, Nativen C Quellcode als Bibliothek nutzen zu können. Dafür bietet Android ein eigenes Development Kit (das Android Native Development Kit - NDK), welches die Cross-Kompilierung auf die Zielplattform durchführt. Auf den meisten Android-Geräten und dem Android-Emulator wird eine ARM Plattform[3] genutzt. Da Android auch eine eigene Bionic genannte *libc* einsetzt, kann so auf einfachem Wege die Übersetzung stattfinden. Die erstellte Bibliothek kann mittels dem Java Native Interface (JNI) eingebunden und genutzt werden.

Dazu müssen innerhalb einer Java-Klasse die Methoden, welche innerhalb der Bibliothek implementiert werden sollen, mit dem Schlüsselwort *native* ausgezeichet werden. Weiterhin muss dafür gesorgt werden, dass sich die Bibliothek im *java.library.path* befindet. Als Beispiel für die Benutzung von JNI kann eine Datei Jni.java erstellt werden:

```
public class Jni {
        public native void sayHello(String name);
}
```

Nachdem diese kompiliert wurde, wird mit dem von Java bereitgestellten Werkzeug *javah* aus der Klasse eine c oder c++ Header-Datei erzeugt. Für das oben angegebene Beispiel sieht die resultierende Header-Datei wie folgt aus:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Jni */

#ifndef _Included_Jni
#define _Included_Jni
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Jni
 * Method:    sayHello
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_Jni_sayHello
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

Diese Datei kann anschließend durch eine entsprechende c oder c++ Quellcode-Datei mit einer Implementierung versehen werden. JNI definiert diverse Datentypen, welche die Standard-Java-Objekte in der nativen Sprache abbilden. Darunter sind beispielsweise die Strukturen `jobject`, `jclass` and `jstring` [14].

[3] http://www.arm.com/

Dieses Vorgehen muss auf der Android-Plattform analog angewendet werden. Zuerst wird die Java-Schnittstelle der Bibliothek definiert, anschließend wird diese Datei in eine Header-Datei umgewandelt und dann durch nativen Code implementiert.

6.5.4. Native Development Kit

Das Kompilieren wird, wie beschrieben, von dem Android NDK durchgeführt. Dazu müssen für das NDK bestimmte Vorbereitungen getroffen werden.

Das NDK wird von dem SDK unabhängig installiert und befindet sich daher in einem eigenen Verzeichnis. Die erzeugten Anwendungsteile hängen allerdings voneinander ab, da die in Java erzeugten Klassen über JNI auf die in c erzeugten Funktionen zugreifen müssen. Daher müssen die nativen Quellcode-Dateien in der SDK Verzeichnisstruktur liegen. Android bestimmt dazu ein Unterverzeichnis in der Projektwurzel, in welches die für JNI benötigten Dateien abgelegt werden. Das dieses Verzeichnis berücksichtigt werden muss, ist dem NDK über eine Konfigurationsdatei mitzuteilen. Im NDK-Verzeichnis existiert ein Unterverzeichnis für jedes Projekt. In diesem muss sich eine *Application.xml* befinden, welche bestimmt, wie das zu kompilierende Modul heißt und wo es sich befindet.

Weiterhin wird ein Makefile benötigt, welches alle Kompilierungsoptionen und Befehle beinhaltet. Dieses wird *Android.mk* genannt und muss sich im *jni*-Unterverzeichnis des Projekts befinden.

Nach einem Aufruf von `make APP=<PROJEKTNAME>` wird die entsprechende Bibliothek anhand der Regeln des Makefiles erstellt. Diese kann anschließend aus dem SDK heraus genutzt werden und wird automatisch auf dem Handy geladen.

Diese Funktion ist vor allem dann nützlich, wenn ein Prozess auf Ressourcen der unteren Schichten des Android Stacks zugreifen will. Solche Ressourcen lassen sich durch die Framework Methoden nicht direkt ansprechen. Die Nutzung einer nativen Bibliothek hingegen ermöglicht den direkten Zugriff, solange die Benutzerrechte des Linux Systems dies erlauben.

6.5.5. Dalvik Debug Monitor Server

Android bietet innerhalb des SDKs eine das normale Eclipse Debugging erweiternde Lösung an. Diese heisst Dalvik Debug Monitor Server (DDMS) und ermöglicht es beispielsweise, die von dem von Android verwendeten Logging-Framework *logcat* erstellten Logs anzuzeigen. Diese lassen sich mit DDMS auch auf bestimmte Quellen filtern. Weiterhin erlaubt es, Systemdaten, wie CPU und Speicherauslastung, oder laufende Threads anzuzeigen. Es ist zusätzlich möglich, den Status des Emulators oder Gerätes zu ändern oder einen Telefonanruf zu simulieren, um die Funktion einer Anwendung zu testen.

6.5.6. Android Debugging Bridge

Die Android Debugging Bridge (ADB) ist ein Werkzeug, welches für die Interaktion mit dem Emulator oder einem mobilen Gerät genutzt werden kann. Es bietet viele verschiedene Funktionen, die den Benutzer oder Entwickler dabei unterstützen. Darunter sind Mechanismen für die Dateiübertragung sowie für die direkte Manipulation der Plattform.

Für die Übertragung von und auf das Gerät können `adb pull` und `adb push` verwendet werden. Da durch das Kopieren eines Paketes dieses nicht installiert wird, gibt es die Funktion `adb install`. Diese interagiert mit dem Installationsprozess und registriert die Anwendung im System. Dies führt zu einem Problem, da eine solche Installation nicht durch den lokalen Verifizierer erkannt und verhindert werden kann. Um dies zu erkennen, müsste eine Remote Attestation als zusätzlicher Schutzmechanismus aktiviert werden oder ein Abgleich der Anwendungen durch ein Repository gegeben sein. Es wird allerdings erkannt, wenn eine solche installierte „Angriffsanwendung“ andere Anwendungen manipuliert.

Eine weitere, sehr wichtige Funktion der ADB ist `adb shell`. Sie öffnet eine Konsole auf dem verbundenen Gerät oder Emulator. So ist ein Debugging und Steuern des Linux Systems möglich.

Im Normalzustand sind nur wenige Programme im PATH der Konsole enthalten. Dies kann durch den Einsatz eines Werkzeugs behoben werden. Busybox[4] ist ein solches Werkzeug. Es erzeugt abgeschwächte Versionen häufig benutzter Programme und fasst sie in einer ausführbaren Datei zusammen. Diese können beispielsweise in dem Busybox-Verzeichnis abgelegt werden. Durch Modifikation der PATH-Umgebungsvariablen auf dieses Verzeichnis sind in der Shell die meisten Standard-Linux-Kommandos verfügbar. Mit diesen lassen sich beispielsweise die Netzwerkeinstellungen überprüfen.

Der PATH kann entweder nach jedem Start der Shell manuell eingerichtet werden oder in der *init.rc* festgeschrieben werden. So sind nach dem Start die Kommandos verfügbar.

6.6. Deployment

Die implementierten Komponenten müssen in die Android-Plattform integriert werden. Dazu ist es nötig, verschiedene Stellen zu modifizieren. Wie bereits erklärt, werden der Kernel angepasst, Prozesse erzeugt und interne Bausteine von Android verändert. Es gibt daher verschiedene Ansatzpunkte, welche genutzt und verändert werden müssen.

6.6.1. Android Source

Die Android-Quellen[5] können aus einem Git Repository heruntergeladen werden. Der Verzeichnisbaum, welcher geladen wird, ist in Abbildung 6.4 dargestellt. Die Ordner sind verschiedenen Funktionen zugeordnet. Das Verzeichnis *bionic* hält zum Beispiel den Quellcode der angepassten C-Standard-Bibliothek. Die Ordner *dalvik* und *external*

[4] http://www.busybox.net/
[5] http://source.android.com/source/download.html

besonders wichtig. Im ersten befinden sich die Quellen der Dalvik Virtuellen Maschine, im zweiten die externen Bibliotheken und Programme, die mit Android ausgeliefert werden sollen, wie zum Beispiel die OpenSSL-Library oder die Library für *iptables*.

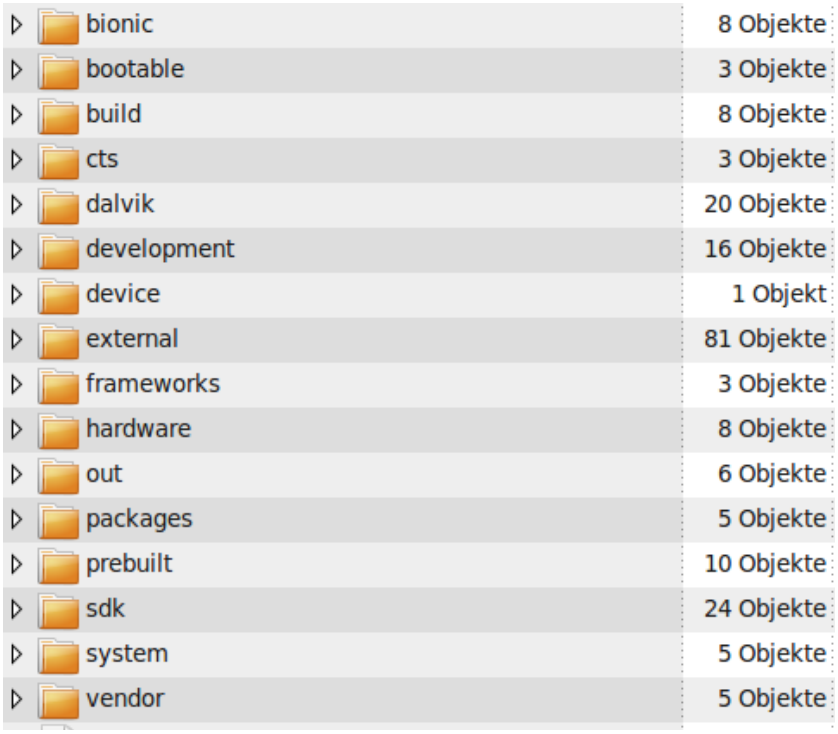

Abbildung 6.4.: Aufbau der Android-Quellen

Android benutzt *make* für die Kompilierung der Quellen, verwendet aber eigene Make-Skripte, welche die Programme beschreiben. Im Anhang A ist das für die Metrik-Bibliothek genutzte Skript dargestellt.

Wie beschrieben wurden mehrere externe Bibliotheken in das *external*-Verzeichnis hinzugefügt. Darunter der Tpm-Emulator, TrouSerS und die Metrik. Um sie in Android integrieren zu können, mussten ebenso entsprechende Makefiles geschrieben werden.

Durch den Build-Prozess von Android werden das *system.img*, das *ramdisk.img* sowie das *userdata.img* erzeugt und in dem Verzeichnis *out/target/product/generic* abgelegt.

6.6.2. Kernel

Android verwendet einen Linux Kernel mit der Version 2.6.29. Dieser ist so angepasst, dass er vom Emulator gestartet werden kann. Für die nötigen Modifikationen mussten die Quellen für den Kernel gepatched werden. So wurde das IMA-Modul, welches ab Version 2.6.30 im Standard-Kernel enthalten ist, zu den Quellen hinzugefügt. Weiterhin mussten an IMA Modifikationen vorgenommen werden (siehe Abschnitt 6.2.2).

Nach dem Patchen wird der Kernel über die *.config*-Datei so angepasst, dass er die IMA-Erweiterungen startet. Die Änderungen zum Erstellen einer Queue benötigen einen eigenen Parameter in der Konfiguration, um das Vorhalten der Messungen einzuschalten. Nachdem der Kernel mit Hilfe der Android Toolchain für die Android-ARM Plattform kompiliert wurde, kann er in der Konfiguration der AVD als Image definiert werden.

Für den Emulator wird ein Kernel verwendet, der mit *QEMU*[6] angepasst ist, ein für die Virtualisierung von Linux-Systemen geeignetes Werkzeug. Dieser ist in der Lage, verschiedenste Plattformen, darunter auch die ARM-Plattform, zu emulieren [2].

Nach dem Kompilieren eines angepassten Kernels ist es möglich, das dabei erstellte Image in dem *images*-Verzeichnis des SDKs abzulegen. Dies wird, nachdem es in *kernel-qemu* umgenannt wurde, automatisch vom Emulator genutzt.

6.6.3. SDK/NDK

Anwendungen, welche mit dem SDK erstellt werden, werden automatisch über den Installationsprozess auf den Emulator geladen. Dies gilt auch für Bibliotheken, welche von dem SDK verwendet werden, besonders auch für über das NDK erstellte native Bibliotheken. Das SDK übernimmt auch das Signieren der Anwendung für den Entwickler, um den Aufwand von Testläufen gering zu halten.

Um eine Anwendung zu vertreiben, muss der Entwickler jedoch mit einem eigenen Schlüssel signieren. Dieser muss nicht von einer CA beglaubigt sein, er sollte sich allerdings nicht ändern, um Updates an Anwendungen zu ermöglichen. Dies ist nur erlaubt, wenn die Signaturen den gleichen Schlüssel besitzen.

[6] http://wiki.qemu.org/Main_Page

7. Tests

Verschiedene Tests sollen zeigen, dass die Funktion der während der Arbeit erzeugten Mechanismen denen des Konzeptes entspricht. Weiterhin soll die Performanz des dadurch erzeugten Gesamtsystems überprüft werden.

7.1. Test des TSS

Da sich der TSS aus vielen Komponenten zusammensetzt, muss die Interaktion der Komponenten getestet werden. Dies wird in mehreren Schritten, die Schichten abbildend, durchgeführt.

7.1.1. TPM-Emulator und TrouSerS

TrouSerS bietet eine Reihe von Konsolenbefehlen, mit welchen die Funktion des TPMs überprüft werden kann. Darunter sind die Befehle `tpm_readpcrs` und `tpm_extendpcrs`. Diese beiden Kommandos werden verwendet, um den TPM über TrouSerS zu testen.

Zuerst wird der Wert des PCRs 10 angezeigt.

```
# tpm_readpcrs 10
10:00000000000000000000000000000000000000000
```

Bei Boot des Systems sind alle PCRs mit 0 vorbelegt. Anschließend wird ein Wert zu dem PCR hinzugefügt, und daraufhin noch einmal überprüft. Die Funktion der Extend-Operation (Siehe Abschnitt 2.3) lässt sich anhand der Konsolenausgaben gut nachvollziehen. Dem Wert von PCR 10 soll der Wert 12345678901234567890 hinzugefügt werden. Dazu werden die beiden Werte verknüpft und das PCR wird neu belegt.

```
# tpm_extendpcrs 10 12345678901234567890
val10:123456789012345678900000000000000000000 new10:
   DAACBBE650AF440EC3230B7FE9E55E6BBC8A1CE8
# tpm_readpcrs 10
10:DAACBBE650AF440EC3230B7FE9E55E6BBC8A1CE8
```

Die Kommunikation von TrouSerS und dem TPM-Emulator ist somit funktionstüchtig. TrouSerS beschreibt die Gerätedatei */dev/tpm*, welche von dem Emulator bereitgestellt wird. Ist diese Datei nicht vorhanden, bricht der Start des TrouSerS TCS-Dienstes mit einer Fehlermeldung ab.

Wird der TPM-Emulator Prozess abgebrochen, während der TCSD noch aktiv ist, reagiert dieser wie im Folgenden dargestellt:

```
# kill 43 // Abbruch des TPM Emulator-Processes
# tpm_readpcrs 10
10:error.........error
```

Fehlt das Geräte-Device schon bei dem Start des TCSD, wird der Start mit einem Fehler abgebrochen.

7.1.2. jTSS

Da die Funktion von TrouSerS gezeigt wurde, wird darauf aufbauend die nächste Schicht getestet. Das jTSS benutzt über JNI die von TrouSerS angebotene Schnittstelle, um auf das TPM zuzugreifen.

Um herauszufinden, ob dieser Zugriff funktioniert, wurde die Android-Anwendung, welche die jTSS-Schnittstelle bereitstellt, so modifiziert, dass der Wert von PCR 10 ausgegeben wurde. Diese Ausgabe ist in Listing 7.1 dargestellt.

Listing 7.1: Ausgabe des jTSS

```
I/getPcrQuote(  233): PCR 10 has value  da ac bb e6 50 af 44 0e c3 23 0
   b 7f e9 e5 5e 6b
I/getPcrQuote(  233):  bc 8a 1c e8
```

Der ausgegebene Wert stimmt mit dem vorher ins TPM geschriebenen Wert überein. Der Zugriff auf das TPM aus Android heraus ist also möglich.

7.1.3. Android-Services

Abschließend ist ein Test der Android Schnittstelle notwendig. Dafür wurde eine spezielle Android-Applikation geschrieben, welche alle Methoden des Services aufruft.

Diese sollte bei einem Aufruf der Methode `getPcrQuote(10)` den in bisher durchgeführten Tests beschriebenen Wert signiert zurückliefern.

Listing 7.2: Ausgabe der Android-Testanwendung

```
I/TssTest (  227): Value of Pcr 10 :
   daacbbe650af44ec323b7fe9e55e6bbc8a1ce8
```

Wie in Listing 7.2 zu sehen, wird der korrekte PCR-Wert als Quote zurückgeliefert. Der Quote wird fehlerfrei als Android-Paket übertragen. Eine den Spezifikationen entsprechende Signatur konnte für den Test nicht erzeugt werden. Dennoch zeigt der Test den vollständigen Durchstich über alle Schichten des Software Stacks: von dem im TPM-Emulator gespeicherten PCR über TrouSerS und den jTSS-Wrapper bis hin zur Android-Schnittstelle und der Nutzung des Services.

7.2. Test des Verifiers

Um den Verifier zu testen, wurde die Dalvik so angepasst, dass Logausgaben erzeugt wurden. In verschiedenen Konstellationen soll so festgestellt werden, ob das System richtig reagiert. Listing 7.3 stellt die Ausgabe dar, welche durch eine erfolgreiche Messung erzeugt wird.

Listing 7.3: Verifikation ohne Eintrag in der Metrik

```
D/dalvikvm(  103): ++++ Dalvik opens: '/system/app/ApplicationsProvider
    .apk'
D/dalvikvm(  103): ########---- verification result is '0' -- Found in
    Metric and OK
D/dalvikvm(  103): MEASUREREQ success
D/dalvikvm(  103): file opened
D/dalvikvm(  103): measurement written
```

Nachdem der Verifier den Wert überprüft hat, wird eine IMA-Messung angestoßen. Kann die Datei nicht in der Metrik gefunden werden, wird die in Listing 7.4 gezeigte Ausgabe erzeugt. Auf diesen Fall wird mit einer Exception reagiert, welche den Start unterbindet. Für den Fall, dass ein etwas weniger restriktives Verhalten gewünscht wird, kann auch auf eine Exception verzichtet werden. Wenn der Start nicht durch die Exception unterbrochen wird, ist ein normaler Programmablauf möglich. Natürlich fehlt bei einer nicht in der Metrik vorhandenen Datei die Sicherheit, dass es sich um die gewünschte Anwendung handelt. Für die Überprüfung und Entwicklung mit einer nicht vollständigen Metrik ist dieser Schritt aber dennoch sinnvoll.

Listing 7.4: Verifikation mit Eintrag in der Metrik

```
D/dalvikvm(  139): ++++ Dalvik opens: '/system/app/Bluetooth.apk'
D/dalvikvm(  139): ########---- verification result is '-1' -- NOT
    Found in Metric
E/AndroidRuntime(  139): Uncaught handler: thread main exiting due to
    uncaught exception
E/AndroidRuntime(  139): java.lang.RuntimeException: Unable to get
    provider com.android.bluetooth.opp.BluetoothOppProvider: java.lang.
    RuntimeException: Package not found in Metric, it is not trusted
```

Nachdem eine Datei durch eine manipulierte Version ausgetauscht wurde, wird die folgende Ausgabe 7.5 erzeugt. Dalvik muss auf den Bibliotheksrückgabewert '1' immer mit einer Exception reagieren, da es sich bei einer Manipulation um einen schwerwiegenden Verstoß handelt. Die Datei hat innerhalb des Zeitraums zwischen Installation und Ausführung ihren Inhalt geändert. Dies ist in keiner denkbaren Konstellation wünschenswert und muss zu einem Programmabbruch führen.

Listing 7.5: Verifikation mit manipuliertem Eintrag in der Metrik

```
D/dalvikvm(  327): ++++ Dalvik opens: '/data/app/de.fhhannover.inform.
    trust.tsse.test.apk'
E/dalvikvm(  327): ########---- verification result is '1' -- File has
    wrong value
D/AndroidRuntime(  327): Shutting down VM
```

```
W/dalvikvm(  327): threadid=3: thread exiting with uncaught exception (
    group=0x4001b168)
E/AndroidRuntime(  327): Uncaught handler: thread main exiting due to
    uncaught exception
E/AndroidRuntime(  327): java.lang.RuntimeException: Unable to
    instantiate activity ComponentInfo{de.fhhannover.inform.trust.tsse.
    test/de.fhhannover.inform.trust.tsse.test.TssServiceTest}: java.
    lang.RuntimeException: Metric Exception: File has wrong value
```

Abschließend wurde vor dem Ausführen einer Verifikation die Metrik entfernt. Als Reaktion darauf soll die Library den Wert '2' zurückgeben und Dalvik soll sowohl eine Log-Nachricht sowie eine Exception erzeugen. Dass dies geschieht, ist in Listing 7.6 zu sehen.

Listing 7.6: Verifikation ohne Zugriff auf die Metrik

```
D/dalvikvm(  281): ++++ Dalvik opens: '/data/app/de.fhhannover.inform.
    trust.tsse.test.apk'
W/dalvikvm(  281): ########---- verification result is '2' -- File can
    not be opend
D/AndroidRuntime(  281): Shutting down VM
W/dalvikvm(  281): threadid=3: thread exiting with uncaught exception (
    group=0x4001b168)
E/AndroidRuntime(  281): Uncaught handler: thread main exiting due to
    uncaught exception
E/AndroidRuntime(  281): java.lang.RuntimeException: Unable to
    instantiate activity ComponentInfo{de.fhhannover.inform.trust.tsse.
    test/de.fhhannover.inform.trust.tsse.test.TssServiceTest}: java.
    lang.RuntimeException: Metric Exception: File can not be opened
```

7.3. Performance

Die Performance des Systems wird vor allem durch die Messungen beeinträchtigt. Wie stark diese Beeinträchtigung ist, soll mit einem Test festgestellt werden.

7.3.1. Boot

Da die meisten Messungen während des Bootvorgangs durchgeführt werden (siehe Abschnitt 6.1), ist wichtig, in welchem Ausmaß dieser verlangsamt ist.

Dazu wurde vor und nach dem Bootvorgang die Systemzeit aufgenommen und mit einem nicht modifizierten System verglichen. Um das Ende festzustellen, wurde die `onCreate`-Methode der PhoneApp modifiziert. Diese ist das Hauptprogramm der Plattform und wird genutzt, um den vollständigen Boot festzustellen. Der eingefügte Code schreibt die Systemzeit in das Log. Diese Zeit, welche in Sekunden ab 1970 angegeben ist, wurde mit der vor dem Start des Emulators gemessenen Zeit verglichen. Daraus ergab sich in zehn Ausführungen für das modifizierte System eine durchschnittliche Ladezeit von etwa 52 Sekunden. Ein nicht modifiziertes System erreichte in der gleichen Anzahl von Läufen eine Zeit von ca. 42 Sekunden.

Der Unterschied ist in den Messungen sowie den zusätzlich gestarteten Komponenten begründet. Diese deutliche Steigerung kann als Nachteil der Architektur gesehen werden. Dennoch ist es auf Android-Geräten selten nötig einen vollständigen Boot durchzuführen, was die Einschränkung reduziert. Dies ist möglich, da die Laufzeit der Akkus im Vergleich zu Notebooks deutlich höher ist. Diese Akkulaufzeit und das Nutzungsverhalten führen bei Smartphones dazu, dass das Gerät immer geladen wird, wenn es nötig ist, ohne es auszuschalten.

7.3.2. IMA

IMA erzeugt Verzögerungen, da besonders bei dem Start des Systems einige Messungen durchgeführt werden. Ein erstellter Test soll zeigen, wie viel Zeit durch die Ausführung einer Messung benötigt wird.

Dabei wurde IMA so erweitert, dass bei dem Einlesen eines Measure-Requests durch die Dalvik VM die Zeitmessung begonnen und nach dem Messvorgang die Zeitmessung abgeschlossen wurde. Daraus resultierte eine Durchschnittsdauer der Messung von etwa 129 Millisekunden. Bei dieser Messung wurden allerdings die Dauer des Schreibens in das virtuelle Dateissystem sowie das Erstellen des Measure-Requests nicht betrachtet. Da der Measure-Request asynchron betrieben wird, war keine vollständige Messung aus der Dalvik VM heraus möglich.

Ein ähnlicher Test wurde von Naumann et. al [16] durchgeführt. Für Ihren, auch auf IMA basierenden und der hier vorgestellten Lösung stark ähnelnden Ansatz, ergab sich für Messungen von Dateien auf dem von ihnen definierten Anwendungslevel eine Messzeit von 1613 Millisekunden. Leider können die Messwerte nicht direkt verglichen werden, da die hier erreichten Messwerte auf einem Emulator durchgeführt wurden, wohingegen die Messungen von Naumann auf einem HTC G1 Mobiltelefon durchgeführt wurden. Weiterhin ist unklar auf welcher Ebene ihre Tests durchgeführt wurden, da die Lösung auch ein angepasstes TPM und einen angepassten TSS verwendet und diese Möglicherweise einen Einfluss auf die Dauer haben könnten.

7.3.3. Lokaler Verifizierer

Für die Geschwindigkeit des Gesamtsystems ist weiterhin wichtig, welchen Zusatzaufwand der lokale Verifizierer erzeugt. Zum Test des Overheads wurde die Anwendung für die manuelle Installation der Metrik mit einer Zeitmessung der Installation und der Verifizierung ausgestattet.

Die Performance der Messungen dauert im Durchschnitt etwa 205 Millisekunden. Die Installation einer Anwendung wird im Vergleich mit der Ausführung nur selten durchgeführt. Daher ist es wichtig, dass besonders die Verifizierung zeitlich begrenzt ist. Der Benutzer sollte keine Veränderung in dem Verhalten des Gerätes bemerken. Sollte der Benutzer bei Start der Anwendung mehrere Sekunden auf diese warten müssen, wäre dies der Fall. Die Verifikation dauerte jedoch im Durchschnitt nur etwa 160 Millisekunden.

Listing 7.7: Ergebnis der Messungen

```
/system/app/GlobalSearch.apk installed in 0.067476 seconds, verified in
    0.059688 seconds
/system/app/CalendarProvider.apk installed in 0.076312 seconds,
   verified in 0.073171 seconds
/system/app/PackageInstaller.apk installed in 0.064323 seconds,
   verified in 0.095677 seconds
/system/app/AlarmClock.apk installed in 0.191961 seconds, verified in
   0.158786 seconds
/system/app/Mms.apk installed in 0.589811 seconds, verified in 0.433593
    seconds
/system/app/LatinIME.apk installed in 0.156500 seconds, verified in
   0.138972 seconds
Directory /system/app installed.
 Average duration is 0.204343 seconds
 Average duration of verification is 0.167342 seconds
```

7.3.4. Benchmarks

Shabtai, Fledel und Elovici[21] testen ihr System mit zwei Benchmarks. Diese Benchmarks testen vor allem die Systemzugriffszeiten, wie beispielsweise die von Lese- und Schreiboperationen.

Ein Benchmark wird auf einem laufenden System gestartet, in dem dieser anschließend Operationen durchführt und deren Zugriffszeit misst. Dies stellt eine Einschränkung dar, da im Gegensatz zu ihrem Ansatz mit dem größten Aufwand der IMA-Messungen während des Bootvorgangs zu rechnen ist. Diese werden von einem Benchmark nicht betrachtet. Da es sich weiterhin um eine im Emulator gestartete Lösung handelt, welche von der Rechenleistung nicht mit denen einer realen Umgebung vergleichbar ist, wird von der Durchführung eigener Benchmarks abgesehen.

8. Abschluss und Fazit

Abschließend werden die erarbeiteten Ergebnisse kritisch betrachtet und Lektionen aufgezeigt, die sich aus dem Ablauf ergeben. Weiterhin werden die offenen Punkte definiert und beschrieben.

8.1. Zusammenfassung

Während der Arbeit wurde ein Konzept aufgestellt, mit dem die Trusted Computing Technologien in die Android-Architektur eingebettet werden können. Dieses Konzept sieht vor, die bekannten und aus konventionellen Systemen etablierten Mechanismen zu übernehmen und weitere, für mobile Plattformen sinnvolle Mechanismen hinzuzufügen.

Es wurden die Personen aufgezeigt, welche ein Interesse an der Sicherheit eines mobilen Gerätes haben und daraus die für sie nötigen Sicherheitsanforderungen geschlossen. Anschließend wurden Lösungen vorgestellt, welche diese Sicherheitsanforderungen erfüllen können.

Ein weiterer großer Teil des Konzeptes befasst sich mit der Integration in die Android-Plattform. Dieses technische Konzept sieht verschiedene, miteinander interagierende Komponenten vor, welche auf den unterschiedlichen Ebenen der Android-Architektur aktiv werden. Für die definierten Aufgaben wurden Alternativen abgewogen und eine entsprechende Komponente ausgewählt.

Darauf basierend wurde die Machbarkeit des Konzeptes anhand eines Prototypen dargestellt. Dieser Prototyp stellt viele der Funktionen bereit, welche in dem Konzept beschrieben wurden.

Abschießend wurden verschiedene Tests durchgeführt, um zu untermauern, dass die Funktionen der erdachten Konzepte die gewünschte Wirkung zeigen. Weiterhin wurden die Geschwindigkeitseinschnitte, welche sich durch die Erweiterungen ergeben, betrachtet und bewertet.

8.2. Bewertung

Das Konzept beschreibt die wichtigsten Ansätze, welche im Trusted Computing genutzt werden. Für diese wird eine Einbettung in die Architektur vorbereitet. Technische Einschränkungen verhindern jedoch ein kompletteres technisches Konzept sowie einen vollständigeren Prototypen. Diese Einschränkungen werden allerdings in Zukunft wegfallen. Ein Mobiltelefon mit einem TPM oder einem MTM auszustatten, sollte keine große Herausforderung darstellen, sobald die MTM-Spezifikation eine auslieferbare Reife erreicht.

Wenn dies passiert, lassen sich Secure und Authenticated Boot, Remote Attestation sowie Local Verification komplett abbilden. Auch leichte Schwächen lassen sich dann durch die Funktionen des TPM/MTMs ausgleichen.

Der anhand des Konzeptes entwickelte Prototyp für die Integration der Trusted Computing Technologien ermöglicht den Zugriff auf die definierten Komponenten und erlaubt so die Durchführung einer Remote Attestation für die Plattform. Weiterhin wird ein Mechanismus geschaffen, welcher die Verifizierung von Anwendungen gestattet, auch wenn keine Netzwerkverbindung verfügbar ist. Diese lokale Verifizierung ist dazu geeignet, den Start von nicht vertrauenswürdiger Software zu unterbinden.

Durch die Nutzung bestehender, etablierter Software sowie der Orientierung an den Standards der Trusted Computing Group ergibt sich ein System, welches den Anforderungen des Trusted Computing gewachsen ist. Dennoch bleiben verschiedene Punkte in der Implementierung des Prototyps offen, welche eine sofortige Nutzung einschränken. Diese werden in Abschnitt 8.4 noch einmal eingehend beschrieben.

Wie die Tests (siehe Kapitel 7) zeigen, lassen sich die vorgestellten Mechanismen einsetzen und profitieren von der Schichten-Architektur auf Android-Plattformen. Besonders die eingeschränkte Möglichkeit, Anwendungen auf einer höheren Schicht nur durch eine Komponente zu starten, ermöglicht die genauere Kontrolle über den Zustand dieser Anwendungen.

Da die gestarteten Anwendungen gemessen werden können, ist im Gegensatz zu herkömmlichen Architekturen eine fast vollständige Abdeckung der Plattform anhand von Integritätsmessungen möglich. Dies ermöglicht eine viel feinere Attestierung und somit mehr Sicherheit für alle Beteiligten.

Die Geschwindigkeitsmessungen zeigen deutliche Einschnitte, dennoch liegen diese vor allem in der Zeit des Bootvorgangs. Da mobile Geräte aufgrund der vorhandenen Akkus recht selten booten, sind zeitliche Einschnitte an dieser Stelle zu verkraften.

Die während der Laufzeit des Geräts auftretenden Einschnitte sind in einem vertretbaren Rahmen, besonders da nur wenige Messungen während des aktiven Betriebs durchgeführt werden.

8.3. Lessons Learned

Neben der Einarbeitung in viele unbekannte Themen bot die Arbeit viele Herausforderungen.

Dabei stellte sich heraus, dass die generelle Arbeitsweise einem Bottom-Up Ansatz folgte. Dies hat sich für die Erstellung des Konzeptes als hinderlich erwiesen, da zuerst die technische Machbarkeit erprobt und anschließend an den Anwendungsfällen gearbeitet wurde. Zusätzlich machte sich diese Vorgehensweise bei der Begründung mancher Zusammenhänge bemerkbar, was sich als problematisch erwies.

Weiterhin ergab sich eine sehr gute Zusammenarbeit mit dem Betreuerteam, die in allen Belangen sehr hilfsbereit und kompetent mitarbeiteten. Diese Teamarbeit ermöglichte ein sehr schnelles Vorankommen.

8.4. Weitergehende Arbeiten

- Der Secure- oder Authenticated-Boot Mechanismus muss implementiert werden, sobald Hardwareunterstützung zur Verfügung steht. Weiterhin muss die Chain of Trust um die dazukommenden Komponenten erweitert werden. Dazu gehören vor allem eine hardwarenahe RTM sowie der Bootloader, welcher die Messung des Kernels durchführen kann.

- Für die Remote Attestation innerhalb der Dalvik VM wird nur ein Bruchteil der von der TCG spezifizierten Schnittstelle angeboten. Dies erfordert noch einen großen Einsatz in der Implementierung, bietet aber das Konzept betreffend keinen Mehrwert. Weiterhin müssen in diesem Schritt die Fehler mit der Nutzung des SRK sowie der Anzeige des EventLogs behoben werden.

- Die lokale Verifizierung erfordert speziellere Schutzmechanismen, welche für den Prototyp nicht aktiviert wurden. Der Einsatz einer Mandatory Access Control-Lösung bietet sich hier an. Auch die Mechanismen, welche durch ein Hardware-TPM zur Verfügung stehen würden, können zur Verbesserung dieser Lösung beitragen.

- Für den von der TCG definierten Standard für Mobile Plattformen in der Version 1.0 gibt es noch keine Realisierung. Daher wurde in dieser Arbeit auf die Einbettung der in dem Standart definierten Elemente verzichtet und auf ein TPM zurückgegriffen. Die von dieser Spezifikation beschriebenen Mechanismen wie RIM Zertifikate und eine Root of Trust for Verification können mit dem lokalen Verifizierer vereinbart werden. Wie im Konzept (siehe Abschnitt 6.4.3) beschrieben, ist eine Nutzung des Verifizierers als RIM_Auth möglich. Die weitere Entwicklung der Spezifikation sollte allerdings abgewartet werden, um zu sehen, ob diese Verfahren weiterhin mit der hier vorgestellten Lösung kompatibel sind.

Literaturverzeichnis

[1] *Android Developer homepage*. http://developer.android.com, May 2010

[2] *QEMU Dokumentation*. http://wiki.qemu.org/download/qemu-doc.html, Juni 2010

[3] ARM: *ARM Security Technology Building a Secure System using TrustZone Technology*. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/CACGCHFE.html, 2010

[4] Azab, A.M. ; Ning, Peng ; Sezer, E.C. ; Zhang, Xiaolan: HIMA: A Hypervisor-Based Integrity Measurement Agent. In: *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, 2009. – ISSN 1063–9527, S. 461 –470

[5] Becker, Arno ; Pant, Marcus: *Android 2. Grundlagen und Programmierung*. dpunkt.verlag, 2010. – ISBN 978–3–89864–677–2

[6] Bishop, Matt ; Bishop, Matt ; Dilger, Michael ; Dilger, Michael: Checking for Race Conditions in File Accesses. In: *Computing Systems* 9 (1996), S. 131–152

[7] Bornstein, Dan: *Dalvik Internals*. http://www.youtube.com/watch?v=ptjedOZEXPM, June 2008

[8] Brickell, Ernie ; Camenisch, Jan ; Chen, Liqun: Direct anonymous attestation. In: *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*. New York, NY, USA : ACM, 2004. – ISBN 1–58113–961–6, S. 132–145

[9] Dietrich, K. ; Winter, J.: Secure Boot Revisited. In: *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, 2008, S. 2360 –2365

[10] Gallery, Eimear: An overview of trusted computing technology. In: Mitchell, C. J. (Hrsg.): *Trusted Computing*. IEE, London, 2005, Kapitel 3, S. 28–114

[11] Grawrock, David: *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2009. – ISBN 1934053171, 9781934053171

[12] IEEE: *IEEE Standard for Local and Metropolitan Area Networks – Port-Based Network Access Control*. http://standards.ieee.org/getieee802/download/802.1X-2004.pdf, 2004

[13] Löhr, H. ; Sadeghi, A.-R. ; Winandy, M.: Patterns for Secure Boot and Secure Storage in Computer Systems. In: *Availability, Reliability, and Security, 2010. ARES '10 International Conference on*, 2010, S. 569 –573

[14] Microsystems, SUN: *JNI Types and Data Structures*. http://java.sun.com/javase/6/docs/technotes/guides/jni/spec-1.4/types.html, 2010

[15] Mitchell, C. J.: What is trusted computing. In: Mitchell, C. J. (Hrsg.): *Trusted Computing*. IEE, London, 2005, Kapitel 1, S. 1–10

[16] Nauman, Mohammad ; Khan, Sohail ; Zhang, Xinwen ; Seifert, Jean-Pierre: *Beyond Kernel-level Integrity Measurement: Enabling Remote Attestation for the Android Platform*. http://www.list.gmu.edu/zhang/, 2010. – to appear in 3rd International Conference on the 3rd International Conference on Trust and Trustworthy Computing (TRUST)

[17] Oberheide, Jon: *remote-kill-and-install-on-google-android*. http://jon.oberheide.org/blog/2010/06/25/remote-kill-and-install-on-google-android/, juni 2010

[18] Pashalidis, Andreas: *Interdomain User Authentication and Privacy*, University of London, Diss., 2005

[19] Sailer, Reiner ; Zhang, Xiaolan ; Jaeger, Trent ; Doorn, Leendert van: Design and implementation of a TCG-based integrity measurement architecture. In: *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*. Berkeley, CA, USA : USENIX Association, 2004, S. 16–16

[20] Schneier, Bruce: *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. New York, NY, USA : John Wiley & Sons, Inc., 1995. – ISBN 0–471–11709–9

[21] Shabtai, A ; Fledel, Y ; Elovici, Y: Securing Android-Powered Mobile Devices Using SELinux. In: *Security Privacy, IEEE* PP (2009), Nr. 99, S. 1–1. http://dx.doi.org/10.1109/MSP.2009.144. – DOI 10.1109/MSP.2009.144. – ISSN 1540–7993

[22] Trusted Computing Group: *TCG Software Stack (TSS) Specification*. http://www.trustedcomputinggroup.org/developers/software_stack/specifications, march 2007. – version 1.2

[23] Trusted Computing Group: *TCG Specification Architecture Overview*. http://www.trustedcomputinggroup.org/files/resource_files/AC652DE1-1D09-3519-ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf, august 2007. – revision 1.4

[24] Trusted Computing Group: *TPM Main Part 1 Design Principles*. http://www.trustedcomputinggroup.org/developers/trusted_platform_module/specifications, july 2007. – version 1.2 Level 2 revision 103

[25] Trusted Computing Group: *Mobile Trusted Module Specification*. http://www.trustedcomputinggroup.org/developers/mobile, june 2008. – version 1 revision 6

[26] Trusted Computing Group: *TNC Architecture for Interoperability*. http://www.trustedcomputinggroup.org/resources/tnc_architecture_for_interoperability_specification, may 2009. – revision 1.4 revision 4

[27] Wang, Xiaoyun ; Yin, Yiqun L. ; Yu, Hongbo: *Finding Collisions in the Full SHA-1*. Bd. 3621. 2005. – 17–36 S. http://dx.doi.org/10.1007/11535218_2

[28] Winter, Johannes: Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In: *STC '08: Proceedings of the 3rd ACM workshop on Scalable trusted computing*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–295–5, S. 21–30

[29] YaPing, Chi ; Lei, Ju ; XiaoDong, Shen ; Yong, Fang: An Improved Sealing Scheme for Trusted Storage. In: *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, 2009, S. 1 –4

Abkürzungsverzeichnis

ADB	Android Debugging Bridge
AIDL	Android Interface Definition Language
AIK	Attestation Identity Key
AIK CA	AIK Certificate Authority
APK	Android Package
AVD	Android Virtual Device
DAA	Direct Anonymous Attestation
DDMS	Dalvik Debug Monitor Server
DEX	Dalvik Executable
EK	Endorsement Key
IMA	Integrity Measurement Architecture
IMC	Integrity Measurement Collector
IMV	Integrity Measurement Verifier
JNI	Java Native Interface
MAC	Mandatory Access Control
MTM	Mobile Trusted Module
NDK	Native Development Kit
PCR	Platform Configuration Register
RIM	Reference Integrity Metric
RTM	Root of Trust for Measurement
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
RTV	Root of Trust for Verification
SDK	Software Development Kit
SML	Stored Measurement Log
SRK	Storage Root Key
TCG	Trusted Computing Group
TCS	TCG Core Services
TDDL	TCG Device Driver Library
TNC	Trusted Network Connect
TOCTTOU	Time of check to time of use
TPM	Trusted Platform Module
TSP	TCG Service Provider
TSS	TCG Software Stack
VK	Verification Key
VM	Virtuelle Maschine

Tabellenverzeichnis

Abbildungsverzeichnis

A. Makefile

Listing A.1 zeigt, wie ein von Android genutztes Makefile aufgebaut ist. Dabei werden verschieden Skripte verwendet, wie das am Anfang des Makefiles definierte `include $(CLEAR_VARS)`, mit dem Variablen zurückgesetzt werden. Mit der Variablen `LOCAL_MODULE` wird der Name der erzeugten Einheit definiert, in diesem Fall die Bibliothek *metircd* und das Programm *metricd_installer*. Anschließend werden die genutzten Bibliotheken und Quelldateien in Variablen festgelegt. Durch `include $(BUILD_STATIC_LIBRARY)` wird schließlich eine statische Bibliothek erzeugt.

Listing A.1: Makefile der Metrik-Bibliothek

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := metricd

LOCAL_SHARED_LIBRARIES := libssl

LOCAL_SRC_FILES := \
 metricd.c \
 src/date.c \
 src/byte_stream.c \
 src/file.c \
 src/map.c \
 src/numbers.c \
 src/string.c \
 src/list.c \
 src/for_each.c \
 src/array.c \
 src/iterator.c \
 src/stack.c \
 src/math_functions.c

include $(BUILD_STATIC_LIBRARY)

include $(CLEAR_VARS)

LOCAL_MODULE := metricd_installer

LOCAL_SHARED_LIBRARIES := libssl
LOCAL_STATIC_LIBRARIES := metricd

LOCAL_SRC_FILES := \
 metricd_installer.c

include $(BUILD_EXECUTABLE)
```

B. CD

Um die mit dieser CD mitgelieferten Emulator-Images ausführen zu können sind die folgenden Dinge zu beachten:

- Ein Android SDK ist installiert
- Das tools/ Verzeichnis des SDKs ist in den PATH eingebunden

Mit `./run_emulator.sh` oder `run_emulator.bat` wird der Emulator mit den vorbereiteten Images ausgeführt. Diese Befinden sich in dem Ordner `android-images`. Nach dem Start kann über die Oberfläche des Emulators mit dem Android System gearbeitet werden. Es lässt sich beispielsweise mit der TssServiceTest-Applikation der Software Stack testen.

Aus einer Shell kann mit `adb shell` auf das Android System zugegriffen werden. Dort lassen sich die Konfigurationen und die IMA-Messungen einsehen. Um einen reibungslosen Start des TPM-Emulators zu gewährleisten, ist es nötig diesen vor Beendigung des Emulators mit kill zu beenden. Sollte dies nicht geschehen, muss die Datei `/data /tpmd_socket:0` manuell gelöscht werden. Weiterhin können mit `adb install` die in dem Unterverzeichnis `apps` liegenden Anwendungen installiert werden.