IPVS

Bachelorarbeit Nr. 113

# Der PMP Gatekeeper

Diana Salsa

| | |
|---|---|
| **Studiengang:** | Informatik |
| **Prüfer/in:** | Prof. Dr. Bernhard Mitschang |
| **Betreuer/in:** | Dipl.-Inf. Christoph Stach |
| **Beginn am:** | 19. Februar 2014 |
| **Beendet am:** | 19. August 2014 |
| **CR-Nummer:** | K.4.1 |

## Kurzfassung

Mobile Endgeräte speichern heutzutage eine große Menge persönlicher Informationen. Verschiedene Betriebssysteme erlauben ein unterschiedliches Maß an Einflussnahme, wobei Android einen großen Teil der Verantwortung an den Anwender abgibt. Das Berechtigungsmanagement des Android-Betriebssystems wird den Anforderungen an Sicherheit und Datenschutz allerdings nicht gerecht. Anwender haben keine Kontrolle darüber, was installierte Apps mit ihren Daten machen. Die Privacy Management Platform (PMP) ermöglicht eine flexible Steuerung der Berechtigungen kompatibler Apps, jedoch werden klassische Apps entweder ignoriert oder vollständig blockiert.

Im Rahmen dieser Arbeit werden verschiedene alternative Berechtigungssysteme analysiert und darauf basierend mögliche Konzepte für die Entwicklung einer ”Gatekeeper”-Komponente für die PMP diskutiert. Diese soll dem Anwender ermöglichen, selbst festzulegen, welche Berechtigungen klassischer Apps erlaubt bzw. blockiert werden sollen. Der Gatekeeper wird als Teil der bereits vorhandenen PMP prototypisch implementiert.

## Abstract

Today’s mobile devices contain a lot of private information. Different operating systems allow a varying degree of user influence; Android for instance passes responsibilities in large part to its users. The Android Permission System cannot, however, adequately satisfy security and data protection requirements. Users have no control over what installed apps do with their data. The Privacy Management Platform (PMP) allows a flexible permission management of compatible apps; however, conventional apps are being either ignored or completely blocked.

This paper analyses different alternative permission systems and discusses possible approaches to implement a PMP ”Gatekeeper” based on that study. This component enables users to decide individually, if permissions of conventional apps are to be blocked or not. The Gatekeeper is prototypically implemented as a part of the existing PMP system.

# Inhaltsverzeichnis

# Abbildungsverzeichnis



# Tabellenverzeichnis

# Verzeichnis der Listings

# 1 Einleitung

In der heutigen Zeit sind Mobiltelefone mehr als nur einfache Handys. Die so genannten ”Smartphones” bieten inzwischen beinahe alles, was auch klassische Computer können und mehr: Verwaltung von Terminen und Kontakten, Zugriff auf Mails und das World Wide Web, Navigation und sogar Bezahlen an der Kasse sind damit heutzutage kein Problem mehr.

Ein Grund für die in den letzten Jahren so rasant gewachsene Beliebtheit von Smartphones ist die große Anzahl verfügbarer Anwendungen (Apps) von Drittanbietern, die zu geringen Preisen und häufig sogar völlig kostenlos bei verschiedenen Quellen (AppStores) erhältlich sind. Ebenfalls ist die Handhabung im Vergleich zu klassischen Computern sehr einfach und intuitiv, wodurch sich auch unerfahrenere Anwender schnell zurecht finden.

Gerade solchen Anwendern ist aber häufig nicht bewusst, welche Risiken durch den leichtsinnigen Umgang mit diesen Geräten entstehen. Dadurch, dass Smartphones in immer mehr Bereichen des täglichen Lebens zum Einsatz kommen, werden auch mehr und mehr persönliche Daten gespeichert und übertragen.

## 1.1 Ausgangssituation

Während es im Zeitalter klassischer Handys noch eine breite Palette verschiedener mobiler Betriebssysteme gab, geht der Trend seit Beginn des Smartphone-Zeitalters in eine deutlich andere Richtung. Der Markt wird heutzutage vom quelloffenen System *Android* dominiert, das im zweiten Quartal 2014 einem Marktanteil von knapp 85% [Kö14] vorweisen kann. Android wird heutzutage von beinahe allen großen Smartphone-Herstellern verwendet, Ausnahmen sind unter anderem Apple (knapp 12%) und Windows (knapp 3%).

Die Firma *Android Inc.* wurde im Oktober 2003 gegründet und nur zwei Jahre später für 50 Millionen US-Dollar von Google aufgekauft [Erd13]. Ende 2007 wurde unter der Leitung von Google die *Open Handset Alliance* als Zusammenschluss verschiedener Handyhersteller, Mobilfunkbetreiber und sonstiger Hard- und Softwareentwickler gegründet. Die ersten mit Android betriebenen Mobiltelefone kamen bereits ein Jahr später auf den Markt.

Vergleicht man verschiedene mobile Betriebssysteme, fällt deutlich auf, dass Nutzer von Android mehr Freiheiten haben als bei anderen Systemen. Apple beispielsweise erlaubt nur die Installation von Apps aus dem eigenen AppStore, wobei nicht jede App dort zugelassen wird. Bei Android können Apps dagegen aus beliebigen Quellen bezogen werden.

Das bedeutet aber gleichzeitig, dass die Verantwortungspflicht, insbesondere im Bereich Sicherheit und Datenschutz, auf den Anwender übergeht. Damit dieser über mögliche Konsequenzen einer Installation informierte Entscheidungen treffen kann, ist das Android-Berechtigungssystem so konzipiert, dass dem Anwender vor der Installation alle von der App geforderten Rechte angezeigt werden und er diese gewähren muss.

Aufgrund des großen Angebots und den weitreichenden Anwendungsgebieten von Apps werden diese heutzutage in nahezu allen Bereichen des täglichen Lebens verwendet. *vHike* [Sta11, SB11] ist beispielsweise ein Dienst zur Bildung von Fahrgemeinschaften, bei dem verfügbare Fahrer in Echtzeit, basierend auf ihrem Standort, an Mitfahrer vermittelt werden. Um alle Funktionen realisieren zu können, wird Zugriff auf Internet, GPS, Bluetooth und Telefonfunktionen benötigt.

Die App *Candy Castle* [SS12] dagegen ist ein auf Google Maps basierendes Lernspiel für diabeteskranke Kinder. Über die Eingabe von gemessenen Blutzuckerwerten werden am jeweils aktuellen Standort Verteidigungstürme für das eigene Schloss gebaut, wodurch Kindern die Wichtigkeit von regelmäßigen Messungen vermittelt wird; gleichzeitig werden Ärzte bei auffälligen Werten automatisch informiert. Die App speichert und übermittelt damit neben Standortinformationen auch medizinische Daten.

Jede App, die wie die vorhergehenden Beispiele sowohl das Internet verwenden darf als auch Zugriff auf personenbezogene Daten hat, könnte diese Informationen potentiell missbrauchen. Der Anwender selbst kann ohne Eingriff in das System allerdings nicht überprüfen, welche dieser Daten tatsächlich versendet werden und an wen.

Dies ist im Normalfall nicht möglich, da Anwender auf Mobilgeräten im Gegensatz zu klassischen Computern standardmäßig keine Administratorrechte haben. Um diese zu erhalten, gibt es für erfahrene Nutzer nur die Möglichkeit, das Gerät zu ”rooten”, dadurch gehen allerdings sämtliche Garantieansprüche verloren.

In der Praxis hat sich daher gezeigt, dass das bestehende Android-Berechtigungssystem die Anforderungen von Anwendern in Bezug auf Sicherheit und Schutz persönlicher Informationen nicht ausreichend erfüllt. Dies liegt insbesondere daran, dass Apps heutzutage eine große Anzahl von Berechtigungen anfordern, wodurch die Privatsphäre von Anwendern immer stärker gefährdet wird [FHE+12].

Aus diesem Grund entstanden in den letzten Jahren verschiedene Konzepte dafür, wie das Berechtigungsmanagement von Android ergänzt werden kann, um mehr Sicherheit zu gewährleisten. Eines dieser Systeme ist die an der Universität Stuttgart entwickelte *Privacy Management Platform (PMP)* [SM13], die anstatt des klassischen Android-Berechtigungsmanagements Rechte nicht global an Apps vergibt, sondern auf Basis einzelner vom Anwender ausgewählter Funktionen.

## 1.2 Problemstellung

Die PMP kann hierbei auf verschiedenen Ebenen des Betriebssystem implementiert werden. Apps, die speziell für die Verwendung mit der PMP entwickelt wurden, greifen ausschließlich darüber auf Ressourcen zu. Alle anderen Apps werden - je nach Implementierungsart - entweder ignoriert oder vollständig blockiert, was in der Praxis im Hinblick auf Datenschutz und Benutzbarkeit beides keine akzeptable Lösung darstellt.

Daher muss die PMP um eine zusätzliche Komponente - den so genannten ”Gatekeeper” - ergänzt werden. Dieser soll dem Anwender ermöglichen, individuell festzulegen, welche Berechtigungen einer App tatsächlich gewährt werden sollen. Dazu muss der Gatekeeper zunächst in der Lage sein, klassische Apps und PMP-Apps eindeutig zu identifizieren; außerdem sollen Zugriffe von System-Apps aus Stabilitätsgründen immer erlaubt werden.

Gegenstand dieser Arbeit ist zunächst eine Analyse bestehender alternativer Berechtigungssysteme für Android. Basierend darauf werden verschiedene mögliche Konzepte für den Gatekeeper vorgestellt und verglichen. Die Komponente wird dann als *Proof-of-Concept* auf Basis des bestehenden PMP-Protoyps implementiert und bewertet.

## 1.3 Gliederung

Die Arbeit ist in folgender Weise gegliedert:

**Kapitel 2 – Das Android-Berechtigungssystem** beschreibt, wie das Rechtemanagement unter Android funktioniert. Hierbei wird insbesondere auf Problematiken aus Sicht von Anwendern und Entwicklern eingegangen; außerdem wird kurz der Aufbau des Android-Betriebssystems beschrieben und erklärt, wie Berechtigungschecks auf Systemebene durchgeführt werden.

**Kapitel 3 – Alternative Berechtigungssysteme** stellt verschiedene Ansätze vor, wie das bestehende Berechtigungssystem erweitert werden kann, um mehr Sicherheit und Kontrolle durch den Anwender zu ermöglichen. Dabei werden verschiedene Implementierungsstrategien und Schwerpunkte verglichen und abschließend bewertet.

**Kapitel 4 – Die Privacy Management Platform** beschreibt Aufbau und Funktionsweise der PMP, insbesondere werden verschiedene Strategien für die Integration in das bestehende Android-Betriebssystem sowie deren Vor- und Nachteile diskutiert.

**Kapitel 5 – Der PMP-Gatekeeper** stellt verschiedene Konzepte für die Entwicklung der neuen Gatekeeper-Komponente der PMP vor. Hierbei werden Wege zur Identifizierung von PMP-Apps vorgestellt und Ansätze diskutiert, wie unerwünschte Berechtigungen von Apps blockiert werden können. Schließlich wird der Gatekeeper auf Basis des PMP-Prototyps implementiert.

**Kapitel 6 – Zusammenfassung und Ausblick** fasst die Kernpunkte und das Ergebnis der Arbeit zusammen und stellt einen Ansatz vor, wie das entwickelte System für die Verwendung in der Praxis weiter verbessert werden kann.

# 2 Das Android-Berechtigungssystem

Android arbeitet nach dem *Principle of Least Privilege*, was bedeutet, dass jede App nur die Informationen und Ressourcen verwenden darf, die ihr jeweiliger Verwendungszweck rechtfertigt. Um Zugriff auf geschützte Inhalte oder Hardwarefunktionen (wie z.B. GPS und Kamera) zu bekommen, müssen der App besondere Rechte erteilt werden. Diese Rechte sind auf eine Menge von *Permissions* abgebildet, mit denen der Anwender zusätzlich benötigte Komponenten für eine App freigeben kann [FCH+11].

## 2.1 Aus Anwendersicht

Will ein Anwender eine neue App aus dem AppStore oder mithilfe eines manuell heruntergeladenen *APK (Android Package)* installieren, erhält er zunächst eine Übersicht über alle besonderen Berechtigungen, die die App anfordert. Daraufhin kann er entscheiden, ob er diese gewähren möchte oder nicht. Falls ja, erhält die App diese Zugriffsberechtigungen auf dem jeweiligen Gerät auf Dauer, ein Zurückziehen ist nur noch durch vollständige Deinstallation der App möglich.

Die Darstellung der geforderten Rechte hat sich mit der Zeit immer wieder verändert, in Android 4.3 werden die Berechtigungen beispielsweise in einzelne Haupt- und Unterkategorien unterteilt (siehe Abb. 2.1): Hauptkategorien sind z.B. *Datenschutz* und *Gerätezugriff*, Unterkategorien sind geordnet nach der jeweils angesprochenen Hardware (z.B. WiFi, Mikrofon, GPS). Zusätzlich ist zu jeder Berechtigung ein kurzer Informationstext abrufbar, der erklärt, wofür diese Berechtigung verwendet werden kann.

Zum Zeitpunkt dieser Arbeit bewirkte eine Neuerung bei Google Play, dass die Berechtigung INTERNET dort im allgemeinen Benachrichtigungsfenster überhaupt nicht mehr angezeigt wird, mit der Begründung, dass "Apps heutzutage normalerweise auf das Internet zugreifen" [goo14]. Dies kann bei Anwendern ein falsches Gefühl von Sicherheit erzeugen, da nicht jedem bewusst ist, dass durch andere Berechtigungen freigegebene Daten über das Internet verbreitet werden könnten [FEW12].

Das System wurde in der Theorie so konzipiert, dass jede App nur die Berechtigungen haben soll, die sie auch tatsächlich benötigt, und der Anwender dadurch über mögliche Zugriffe und damit verbundene potentielle Risiken informiert ist. Hierbei sollte der Anwender bei Apps, die eine ungewöhnliche Menge von Berechtigungen erfordern, obwohl deren Funktionalität damit scheinbar nichts zu tun hat, Vorsicht walten lassen.

In der Praxis ist diese Hemmschwelle allerdings nicht mehr besonders effektiv, da heutzutage beinahe jede App eine gewisse Anzahl von besonderen Berechtigungen benötigt. In [BCG13] wird beschrieben, wie Anwender mit der Zeit darauf "trainiert" werden, Warnhinweise automatisch zu bestätigen. Dieses


**Abbildung 2.1:** Berechtigungen, die von ”Google Suche” angefordert werden (Auszug)

Verhalten ist vergleichbar mit dem Akzeptieren einer Endbenutzer-Lizenzvereinbarung (EULA) bei der Installation von neuer Software, ohne diese überhaupt durchzulesen.

Ein weiteres Problem ist, dass auch wenn eine App tatsächlich eine bestimmte Berechtigung für ihre Kernfunktionalität benötigt, keine Garantie besteht, dass diese Zugriffsfreigabe im Hintergrund nicht für andere unerwünschte Zwecke ausgenutzt werden. Viele Apps erfordern die Berechtigung INTERNET, meistens um damit Werbung anzuzeigen, mit der kostenlose Apps i.d.R. finanziert werden. Hat die App aber gleichzeitig Zugriff auf private Daten wie Kalender, Kontakte oder Standortinformationen (was aufgrund ihrer primären Aufgaben durchaus gerechtfertigt sein kann), kann der Anwender nicht ohne weiteres überprüfen, ob möglicherweise private Informationen unerlaubt an Dritte gesendet werden. Die Studie [EGC+10] hat gezeigt, dass eine nicht zu vernachlässigende Anzahl an Apps entsprechende Informationen an Server weiterleitet, was eine deutliche Verletzung der Privatsphäre von Anwendern darstellt.

Ebenfalls problematisch sind Apps, die durch *Privilege Escalation* unberechtigten Zugriff auf geschützte Daten erhalten können [ZXMX13]. Hierbei greift eine App auf eine zweite App zu, die wiederum Zugriff auf private Informationen hat. Dies wird beispielsweise ermöglicht, wenn beide Apps dieselbe Signatur haben, da solche Apps (wie in Kapitel 2.3 beschrieben) grundsätzlich gegenseitigen Dateizugriff haben. Eine weitere Möglichkeit besteht, wenn Apps selbst als *Content Provider*[1] dienen, ohne die Berechtigungen der aufrufenden Apps zu überprüfen.

Für Anwender, die sich dieser Risiken bewusst sind, wäre die einfachste Lösung, keine Apps mit kritischen Berechtigungen zu installieren und sich stattdessen nach Alternativen umzusehen. Allerdings ist dies häufig umständlich, da Apps im AppStores häufig nach Beliebtheit sortiert sind und man keine Möglichkeit hat, nach Apps ohne bestimmte Berechtigungen zu suchen. Bei offiziellen Apps eines bestimmten Anbieters (wie z.B. Twitter oder Facebook) gibt es auch keine tatsächliche Alternative.

[1]Content Provider stellen Schnittstellen bereit, über die Inhalte mit fremden Prozessen/Apps geteilt werden können.

Aufgrund der zuvor beschriebenen Sicherheitsrisiken wäre es daher wünschenswert, wenn bei der Installation von Apps einzelne Berechtigungen verweigert werden könnten. Viele Apps fordern Berechtigungen, die nur für ein einziges besonderes Feature innerhalb der App nötig sind (z.B. bei einem Bildermanager ein Button zum Hochladen eines Bildes auf eine Fotoplattform). Viele Anwender wollen diese Funktion gar nicht verwenden, müssen jedoch trotzdem die dazugehörige Berechtigung INTERNET gewähren und das damit einhergehende Datenschutzrisiko akzeptieren.

Alles in allem ist das Android-Berechtigungssystem aus Anwendersicht sehr einfach gestaltet, allerdings sind sich viele nicht des Risikos bewusst, das sie durch Installation und Nutzung einer App mit bestimmten zusätzlichen Berechtigungen eingehen. Aber auch wenn man vorsichtig sein möchte und sich im AppStore nach Alternativen umschaut, gibt es häufig keine; daher werden letztendlich doch wieder potentiell gefährliche Apps installiert [AZHL12].

## 2.2 Aus Entwicklersicht

Beim Entwickeln einer App für Android müssen alle Berechtigungen, die eine App benötigt, in Form von *<uses-permission>*-Tags im *Manifest*[2] eingetragen werden (siehe Lst. 2.1). Wird eine Methode aufgerufen, ohne dass die App die nötige Berechtigung hat, tritt eine *Exception* auf. Da der Entwickler aber aufgrund des zuvor beschriebenen Installationsvorgangs davon ausgeht, dass beim Ausführen einer App sämtliche geforderten Berechtigungen gewährt wurden, wird diese Exception häufig nicht abgefangen, wodurch sie die App zum Absturz bringen kann.

**Listing 2.1** Eingetragene Berechtigungen im Manifest der Twitter-App (Auszug)

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.READ_PROFILE" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
...
```

Ein Problem, mit dem Entwickler konfrontiert werden, ist, dass es keinerlei vollständige Übersichtsliste von Methoden mit dazugehörigen Permissions gibt, das heißt, es ist nicht immer klar, ob eine Methode besondere Berechtigungen benötigt. Dies ist zunächst einmal nicht schlimm, da man beim Testen der App früh genug bemerkt, wenn Rechte fehlen.

Umgekehrt passiert es aber häufig, dass Entwickler ihren Apps zu viele Rechte geben, da sie meinen, dass diese für die verwendeten Methoden notwendig sind. [FCH+11] zeigt, dass insbesondere die Berechtigungen ACCESS_NETWORK_STATE und READ_PHONE_STATE häufig unnötigerweise von Apps angefordert werden.

[2] Das Manifest einer App enthält sämtliche Metadaten und Informationen über alle enthaltenen Komponenten der App (inklusive bereitgestellte Dienste) sowie verwendete Bibliotheken und sonstige Abhängigkeiten.

Ebenfalls können Rechte über Berechtigungsgruppen vergeben werden, die eine Menge von Funktionen umfassen. INTERNET beispielsweise erlaubt alle Netzwerkzugriffe, es gibt aber keine Möglichkeit, diese nur auf bestimmte Adressen einzuschränken. Hierbei kann es auch passieren, dass Entwickler eine Berechtigungsgruppe verwenden, die eine benötigte Permission enthält, obwohl dieselbe Permission bereits Teil einer anderen (bereits verwendeten) Gruppe ist. Dies führt dazu, dass die zusätzlichen Berechtigungen der neuen Gruppe unnötigerweise angefordert werden, was ein höheres Sicherheitsrisiko darstellt.

Auch werden häufig Designentscheidungen getroffen, die besondere Rechte erfordern, obwohl dies eigentlich gar nicht nötig wäre. So würde es z.B. bei vielen Apps ausreichen, Daten intern zu speichern statt im öffentlichen Dateisystem, was eine spezielle Berechtigung erfordert. Auch wird der Telefonstatus abgefragt, nur um die IMEI-Nummer als eindeutige ID des Geräts zu erhalten (häufig zu Tracking- und Werbezwecken, wobei dies vom Google PlayStore inzwischen verboten wurde [Rud13]). Gefährlich ist dies, da mit derselben Berechtigung auch Informationen über Netzanbieter, Mailbox, Land, aktuellen Datenverbrauch und Telefonate abrufbar sind [Izz14].

Neben den bereits existierenden Berechtigungen können von App-Entwicklern auch neue Permissions definiert werden. Diese enthalten die Elemente *Name*, *Beschreibung*, *Bezeichner*, *Gruppe* und *Icon* sowie einen von vier *protection Levels*: *normal* (wird bei der Installation nicht extra nachgefragt), *dangerous* (muss vom Anwender bestätigt werden), *signature* (wird nur an Apps gewährt, die dieselbe Signatur haben), *signatureOrSystem* (nur für System-Apps oder Apps mit derselben Signatur).

Mithilfe dieser Definitionen können bestimmte öffentliche Funktionen einer App für andere Apps eingeschränkt werden. Hierbei muss die App, die die Definition bereitstellt, zuerst installiert werden, da diese vom System sonst nicht erkannt wird.

Auch durch neue Android-Versionen können komplett neue Permissions hinzukommen, die von diesem Zeitpunkt an für Zugriffe benötigt werden, die in früheren Versionen noch keine besonderen Rechte erfordert haben. Dies kann problematisch sein, wenn Entwickler ihre Apps nicht entsprechend updaten, wodurch diese auf neueren Systemen aufgrund der ”fehlenden” Rechte nicht mehr lauffähig sind. Allerdings kann es auch vorkommen, dass die neuen Permissions nur System-Apps gewährt werden dürfen, wodurch entsprechende Apps auf neueren Android-Versionen überhaupt nicht mehr benutzt werden können [SC13].

## 2.3 Aus Systemsicht

Das Android-Betriebssystem besteht aus mehreren Schichten (siehe Abb. 2.2), die über verschiedene Schnittstellen miteinander kommunizieren können. Installierte Apps befinden sich auf dem *Application Level*. Das *Application Framework* enthält alle Manager-Komponenten, die Funktionen des Betriebssystems steuern. Diese sind auf oberster Ebene in Java programmiert, die tatsächliche Implementierung liegt aber häufig in nativen Bibliotheken.

Die Kernkomponenten jeder App (*Activities, Services, Receivers* und *Providers*) werden indirekt von der durch das System bereitgestellten Klasse *Context* abgeleitet. Diese enthält die Methoden zur Kommunikation mit dem obersten Level des Application-Frameworks, insbesondere werden über sie auch sämtliche Berechtigungschecks zur Laufzeit durchgeführt.

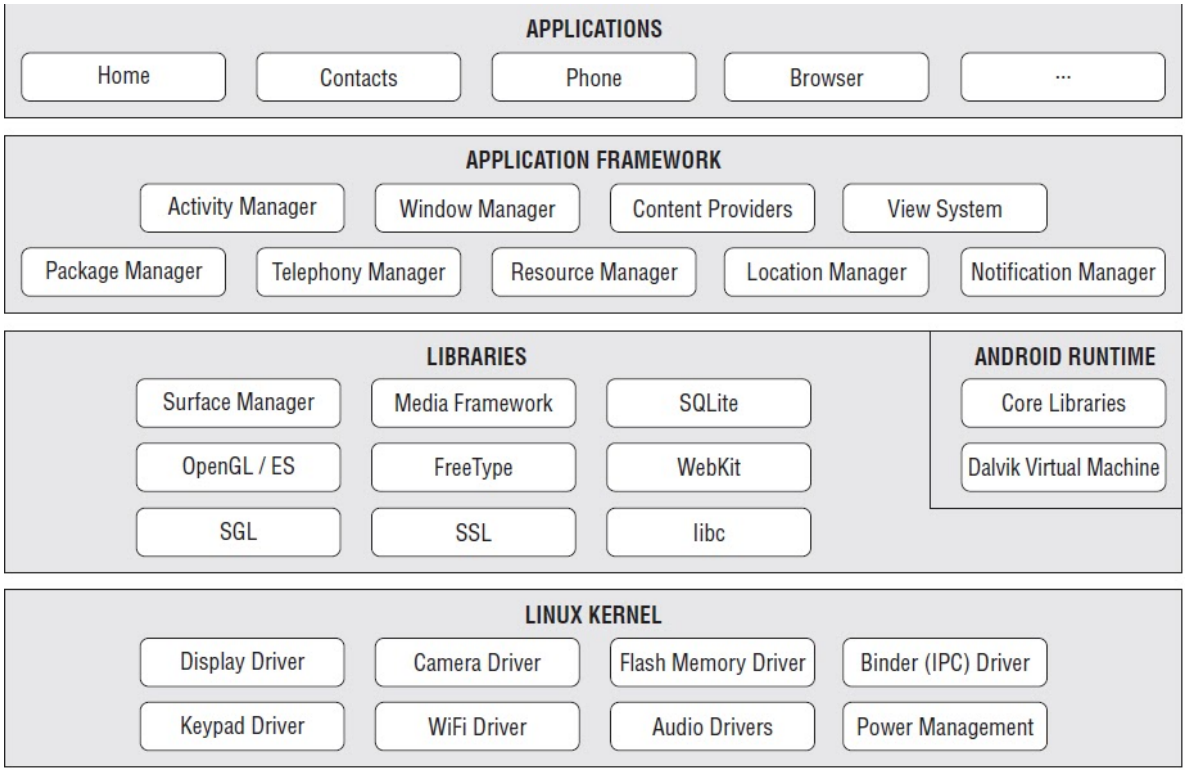



**Abbildung 2.2:** Schichten der Android-Architektur [Kum12]

Der Java-Code der Android-Apps wird in Bytecode übersetzt und innerhalb der so genannten *Dalvik Virtual Machine* ausgeführt. Hierbei handelt es sich um eine Umgebung, die ressourcenschonender ist als die klassische *Java Virtual Machine*, was sie für die Verwendung für mobile Endgeräte geeigneter macht (Hinweis: Zum Zeitpunkt dieser Arbeit wurde in Android 4.4 eine neue experimentelle Laufzeitumgebung namens *ART* eingeführt, die Dalvik langfristig ersetzen soll [sou14]).

Der Kern von Android ist ein Linux-Mehrbenutzersystem, wobei jede App eine eigene Nutzer-ID (UID) erhält. Einzige Ausnahme sind Apps, die von demselben Entwickler stammen und bewusst mit demselben Zertifikat signiert wurden, sowie dieselbe *sharedUserId* haben [dev14].

Für jede UID wird eine separate Instanz der Dalvik VM ausgeführt, wodurch Apps von verschiedenen Entwicklern vollständig voneinander isoliert sind und daher auch keinen gegenseitigen Zugriff auf Daten erhalten können. Einzige Ausnahme sind Dateien, die in öffentlichen Verzeichnissen abgelegt werden, diese können von jeder App mit entsprechenden Rechten gelesen werden.

Ebenso kann der Quellcode anderer Apps sowie deren Kommunikation mit dem Application-Framework nicht direkt manipuliert werden. Allerdings liegen die APKs aller installierter Apps und Dienste in öffentlichen Verzeichnissen, so dass darin enthaltene Informationen von allen Apps gelesen werden können.

Da Apps vollkommen isoliert ausgeführt werden, haben sie zunächst einmal keinen Zugriff auf Funktionen, die auf Systemebene implementiert sind, wie z.B. Zugriff auf die Gerätehardware (Kamera, GPS) oder Netzwerkfunktionen. Hierzu werden spezielle Berechtigungen benötigt, die vom Anwender bei der Installation gewährt werden müssen.

Ruft eine App eine Methode auf, die derartige Rechte erfordert, wird von Context mit *enforce()* ein entsprechender Check ausgeführt. Dieser wird an den ActivityManagerService weitergeleitet, der überprüft, ob die entsprechenden Berechtigungen gewährt wurden. Wenn nicht, wird eine *SecurityException* ausgelöst (siehe Abb. 2.3).

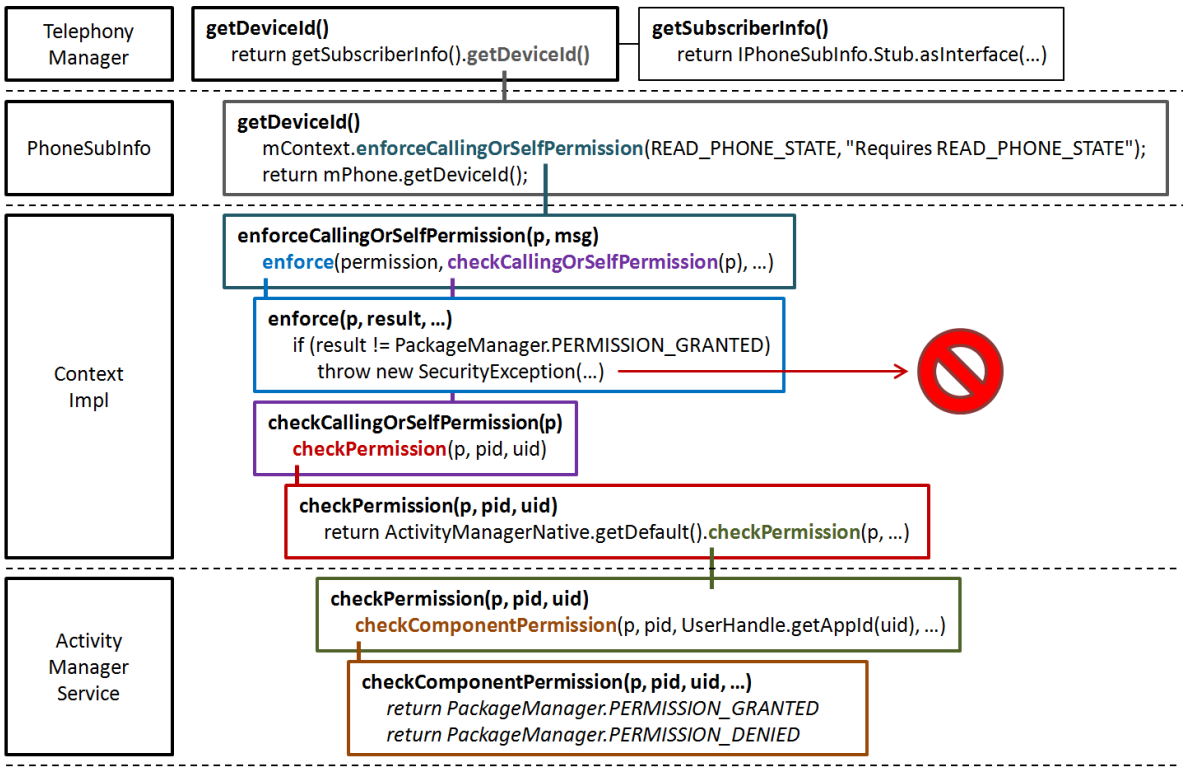



**Abbildung 2.3:** Vereinfachter Ablauf eines Berechtigungschecks beim Aufruf von TelephonyManager.getDeviceId() (auf Basis des Android-Quellcodes)

Der tatsächliche Berechtigungscheck wird von der Methode *checkComponentPermission()* der Klasse *ActivityManagerService* auf Basis von Permission, Prozess-ID und UID der App durchgeführt. Wie genau diese Methode arbeitet, unterscheidet sich jedoch zwischen verschiedenen Android-Versionen.

In Android 2.3 (siehe Lst. 2.2) wird zunächst überprüft, ob es sich um Root- oder System-User handelt, diese erhalten die Berechtigung automatisch, dasselbe gilt für Anfragen aus dem eigenen Prozess. Alle anderen Checks werden über den *PackageManager* abgehandelt, dabei wird mit *checkUidPermission()* aus der UID der App eine Instanz von *GrantedPermissions* ermittelt und überprüft, ob die übergegebene Permission darin enthalten ist, oder nicht.

**Listing 2.2** ActivityManagerService.checkComponentPermission() in Android 2.3 (Auszug) [Thea]

```
// Root, system server and our own process get to do everything.
if (uid == 0 || uid == Process.SYSTEM_UID || pid == MY_PID || !Process.supportsProcesses()) {
      return PackageManager.PERMISSION_GRANTED;
}
// If the target requires a specific UID, always fail for others.
if (reqUid >= 0 && uid != reqUid) {
      Slog.w(TAG, "Permission denied: checkComponentPermission() reqUid=" + reqUid);
      return PackageManager.PERMISSION_DENIED;
}
if (permission == null) {
      return PackageManager.PERMISSION_GRANTED;
}
try {
      return AppGlobals.getPackageManager().checkUidPermission(permission, uid);
} catch (RemoteException e) {
      // Should never happen, but if it does... deny!
      Slog.e(TAG, "PackageManager is dead?!?", e);
}
return PackageManager.PERMISSION_DENIED;
```

Die tatsächlichen Informationen über die Berechtigungen jeder App liegen im Hauptspeicher. Nach Installation oder Update einer App werden die jeweils geforderten Berechtigungen aus dem Manifest gelesen, das als Teil des APKs für jede App im System abgelegt ist, und der Speicher aktualisiert. Dadurch können Berechtigungschecks zur Laufzeit schnell durchgeführt werden. Ebenfalls von Vorteil ist hierbei, dass es keine manipulierbare Datenbasis gibt, durch die böswillige Apps Zugriff auf geschützte Daten erlangen könnten.

## 2.4 Fazit

Das Android-Berechtigungssystem ist zwar vom Grundprinzip her für Anwender leicht verständlich, allerdings hat die in der Praxis immer weiter zunehmende Menge der von Apps geforderten Berechtigungen dazu geführt, dass kaum noch darauf geachtet wird, ob diese Forderungen tatsächlich sinnvoll erscheinen. Aber auch bei Apps, die offensichtlich Zugriff auf private Daten und das Internet benötigen, ist nicht garantiert, dass dies nicht für weitere versteckte Zwecke ausgenutzt wird.

Ein großer Teil der Problematik liegt auch in der fehlenden Möglichkeit, einer App nur ausgewählte Berechtigungen zu gewähren bzw. den Zugriff auf bestimmte Informationen zu unterbinden. In Android 4.3 wurde zwar *App Ops* eingebaut, die genau dies ermöglicht, allerdings ist sie im System versteckt und wurde von manchen Geräteherstellern (z.B. Samsung) wieder vollständig deaktiviert. Ab Android 4.4.2 ist sie sogar nur noch mit Root-Rechten abrufbar [Ros13].

Insgesamt ist das Berechtigungsmanagement in den bisherigen Standardversionen des Android-Betriebssystems nicht ausreichend, um die Sicherheit privater Informationen zu gewährleisten.

# 3 Alternative Berechtigungssysteme

Aufgrund der beschriebenen Unzulänglichkeiten des Android-Berechtigungssystems sind im Laufe der Jahre verschiedene Konzepte entstanden, wie das bestehende System aus Anwendersicht verbessert werden könnte. Im einfachsten Fall wären bereits eine verständlichere Darstellung und Verdeutlichung kritischer Rechtekombinationen hilfreich [SLG+12]. Grundsätzlich liegt das Problem allerdings in der fehlenden Möglichkeit, Berechtigungen nur selektiv zu gewähren.

Zu diesem Zweck wurden verschiedene Systeme entwickelt, die es Anwendern erlauben, mehr Einfluss auf das Berechtigungsmanagement ihrer Apps zu nehmen. Diese lassen sich in zwei grundlegende Kategorien unterteilen: eine Manipulation der Apps durch eine eingebaute Monitor-Komponente oder eine Anpassung des Betriebssystems selbst.

Die hier beschriebenen Verfahren stellen eine Auswahl aus der breiten Menge der existierenden alternativen Berechtigungssysteme dar und sollen verschiedene konzeptionelle und implementierungstechnische Ansätze zeigen.

## 3.1 Inline Reference Monitoring

Da eine App aufgrund des Sandbox-Modells von Android (siehe Kapitel 2.3) nicht in der Lage ist, andere Apps von ”außen” zu überwachen, besteht ein Lösungsansatz darin, existierende Apps durch einen eingebauten Sicherheitsmonitor zu ergänzen. Dieser ist dann in der Lage, Zugriffe abzufangen, die besondere Berechtigungen erfordern, und auf Basis von Richtlinien zu entscheiden, ob diese blockiert oder erlaubt werden sollen.

Um bereits gepackte Apps nachträglich zu bearbeiten, werden Tools verwendet, die zunächst das bestehende APK dekompilieren und damit annäherungsweise die ursprüngliche Datenstruktur wiederherstellen, so dass Änderungen am Quellcode vorgenommen werden können. Danach wird der Code wieder gepackt und neu signiert. Das dadurch entstandene neue APK kann vom Anwender normal installiert werden [HSD13].

### 3.1.1 Dr. Android and Mr. Hide

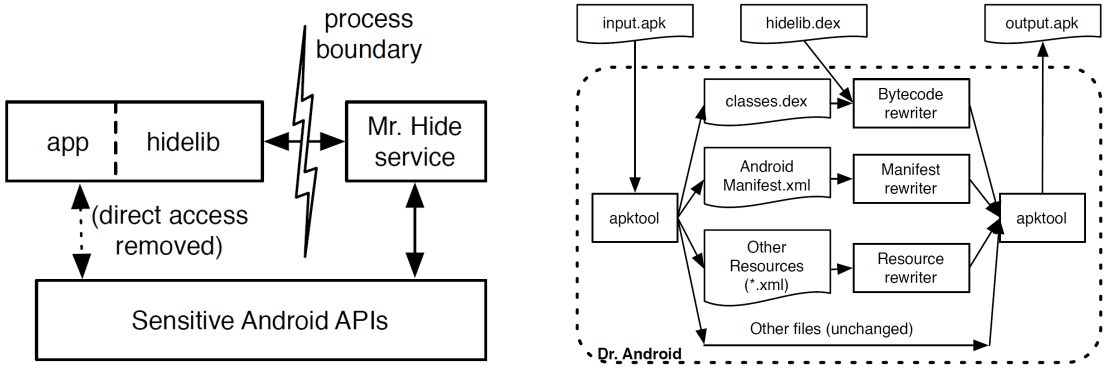
Ein Merkmal vieler Apps ist, dass sie aufgrund der teilweise ”groben” Einteilung der Permission-Elemente mehr Rechte haben, als sie für die Ausführung ihrer Funktionalität tatsächlich benötigen würden. Da mehr Rechte automatisch ein höheres Sicherheitsrisiko darstellen, ist es wünschenswert, diese auf das absolut nötige Minimum zu reduzieren.

Ein Ansatz, der diese Problematik angeht, ist in [JMV+12] beschrieben. Hierbei wurden zunächst die existierenden Android-Berechtigungen anhand ihrer Eigenschaften in vier Hauptgruppen unterteilt: *Externe Ressourcen*, *Strukturierte Anwenderdaten*, *Sensoren* und *Systeminformationen*. Auf Basis der bestehenden Permissions sind neue, stärker eingeschränkte Berechtigungen definiert.

Das Verfahren teilt sich in drei separate Module:

Zunächst wird die existierende App auf notwendige Rechte untersucht. Dies geschieht mithilfe des Tools *RefineDroid* in Form einer statischen Analyse des Dalvik-Bytecodes. Dieser wird auf bestimmte Textmuster und API-Aufrufe geprüft, die Hinweise auf benötigte Berechtigungen geben könnten. Ebenfalls wird ausgewertet, welche Parameter an entsprechende Methoden weitergegeben werden. Daraus ergibt sich eine Liste aller tatsächlich benötigten Rechte der App.

Das Modul *Mr. Hide (the Hide Interface to the Droid Environment)* stellt eine Reihe von Android-Services dar, die eine eigene Menge von Permissions zur Verfügung stellen und regeln (siehe Abb. 3.1a). Damit diese Services die entsprechenden Berechtigungschecks auch abfangen können, müssen Apps, die das System verwenden, passende Schnittstellen enthalten. Diese werden Entwicklern durch die Bibliothek *hidelib* zur Verfügung gestellt. Sie erhalten somit die Möglichkeit, verfeinerte Berechtigungen innerhalb ihrer Apps zu verwenden.




**(a)** Mr. Hide: Bereitstellung verfeinerter Permissions für andere Apps

**(b)** Dr. Android: Ergänzung bestehender Apps durch hidelib-Funktionalität

**Abbildung 3.1:** Dr. Android and Mr. Hide [JMV+12]

Um auch als Anwender eine bereits bestehende App mit den beschriebenen Methoden ergänzen zu können, wird das Modul *Dr. Android* (auf Basis von *apktool*[1]) angeboten (siehe Abb. 3.1b). Dieses entfernt die zu groben Permissions aus dem Manifest und fügt die tatsächlich benötigten eigenen verfeinerten Permissions hinzu. Die Bibliothek hidelib wird so in die App integriert, dass ihre Services beim Starten der App direkt mit ausgeführt werden. Ebenfalls werden Ableitungen von Klassen der *CoreAPI* im Quellcode durch Ableitungen äquivalenter hidelib-Klassen ersetzt, die entsprechende Schnittstellen enthalten.

[1] https://code.google.com/p/android-apktool/

### 3.1.2 I-ARM Droid

Eine Möglichkeit, das Verhalten von Apps beim Aufruf bestimmter Methoden zu manipulieren, bietet *I-ARM Droid* [DSKC12] bzw. dessen Weiterentwicklung *RetroSkeleton* [DC13]. Hierbei geht es zunächst einmal nicht konkret um Funktionalitäten, die besondere Berechtigungen erfordern, jedoch kann das System unter anderem auch dazu verwendet werden, solche Zugriffe abzufangen und mit benutzerdefiniertem Verhalten zu ergänzen.

Um eine Methode zu überschreiben, muss der Anwender deren komplette Signatur angeben. Da diese Signatur innerhalb des Dalvik-Bytecodes einzigartig ist, wird garantiert, dass tatsächlich nur die gewünschte Methode manipuliert wird. Das alternative Verhalten kann in Java-Code formuliert werden, so dass beliebig einfache oder komplexe Anpassungen möglich sind.

Die so entstandenen neuen Methoden werden in separaten Klassen abgelegt, die beim Umschreiben der App hinzugefügt werden. Ursprüngliche Klassen des Application-Frameworks, die neu definierte Methoden enthalten, werden neu abgeleitet und ebenfalls hinzugefügt. Dabei werden die neuen Methoden so ersetzt, dass sie wiederum auf die neuen Definitionen verweisen.

Durch dieses Vorgehen können mehrere Methoden dieselbe Definition haben können, ohne dass ein unnötiger Overhead an Code entsteht. Der ursprüngliche Quellcode der App wird schließlich so umgeschrieben, dass Instanzen von neu abgeleiteten Klassen diese anstatt der Originalklassen verwenden.

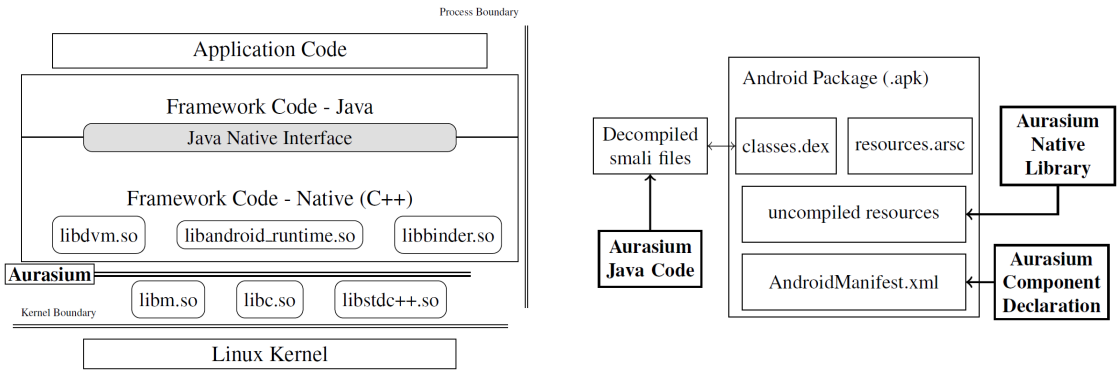
Dieses Verfahren kann verwendet werden, um beliebige Aufrufe von Methoden des Application-Frameworks abzufangen und zu manipulieren. Da dies auch die Methoden beinhaltet, die für Berechtigungschecks verantwortlich sind, können der App an dieser Stelle Berechtigungen ”entzogen” werden. Auch kann bei Aufrufen von Methoden, die private Daten auslesen, stattdessen ein benutzerdefiniertes Objekt (beispielsweise mit Dummy-Daten) zurückgegeben werden.

Dies hat den Vorteil, dass die App trotz effektiv fehlender Berechtigungen nicht abstürzt, da die Methode passende Daten erhält und keinen Fehler wirft. Auch sind anhand der an die Methode übergebenen Parameter kontextbezogene Entscheidungen möglich, beispielsweise könnten Internet-Anfragen nur an bestimmte Server zugelassen werden.

### 3.1.3 Aurasium

*Aurasium* [XSA12] ist ein System, welches eine App um eine zusätzliche Sandbox ergänzt. Dabei wird einer existierenden App eine native Monitor-Komponente hinzugefügt, die bei Anfragen an das System zwischengeschaltet wird und damit in der Lage ist, potentiell unerwünschte Zugriffe auf private Daten abzufangen.

Der oberste Level des Application-Frameworks, also die Interfaces, die von den Apps direkt angesprochen werden, sind in Java programmiert, der Rest in nativem Code. Die tatsächliche Kommunikation wird dort von *Shared Objects* gesteuert, die wiederum auf verschiedenen *Shared Libraries* aufbauen. Dabei landen Anfragen einer bestimmten Funktionalität (z.B. Netzwerkzugriff) - unabhängig davon, welches Interface auf Java-Ebene verwendet wurde - schlussendlich immer bei denselben nativen Methoden. Diese fordern die gewünschte Funktionalität dann vom Linux Kernel an.




**(a)** Architektur von Aurasium

**(b)** Ergänzung des APKs

**Abbildung 3.2:** Aurasium [XSA12]

Aurasium ergänzt die vorhandenen nativen Bibliotheken durch eigene Objekte, die Monitorfunktionen implementieren (siehe Abb. 3.2a). Shared Objects im nativen Teil des Frameworks werden dynamisch mit ihren jeweiligen Zielfunktionen verlinkt. Um den Monitor zwischenzuschalten, werden die verlinkten Funktionszeiger durch Zeiger auf entsprechende Methoden von Aurasium ersetzt, die wiederum auf die tatsächliche Zielmethode zurück verweisen. Dadurch kann der Monitor bei jedem Aufruf entscheiden, ob die Anfrage weitergeleitet oder blockiert wird.

Um diese Entscheidung zu treffen, sind Richtlinien definiert, die für jede angeforderte Funktionalität ein individuelles Verhalten festlegen. So können beispielsweise Netzwerk-Anfragen mit externen Tools analysiert werden, um dem Anwender Informationen über die angefragte Seite zu zeigen.

Alle Zugriffe werden zunächst von Aurasium abgefangen, der Anwender wird dabei jedes Mal informiert und gefragt, ob der jeweilige Zugriff erlaubt werden soll. Dabei ist es auch möglich, individuelle Entscheidungen für die Zukunft zu speichern (z.B. Blacklist/Whitelist für IP-Adressen).

In der einfachen Version wird jede App über einen eigenen Monitor separat gesteuert. Eine Möglichkeit, Zugriffs-Entscheidungen global festzulegen, bietet der *Aurasium Security Manager*. Hierbei werden Anfragen von den Aurasium-Komponenten der verschiedenen Apps immer erst an den Manager weitergeleitet, der die jeweilige Entscheidungen registriert und zurückgibt. Dies hat den Vorteil, dass vom Anwender definierte Regeln auch dann erhalten bleiben, wenn einzelne Apps deinstalliert werden, und auch für neu installierte Apps bereits registriert sind.

Aurasium verwendet *apktool*, um das APK einer bestehenden App zu ergänzen (siehe Abb. 3.2b). Hierbei können die benötigten nativen Bibliotheken als Shared Objects ohne Integrationsaufwand hinzugefügt werden, zusätzlich wird der benötigte Java-Code in den bestehenden Quellcode integriert. Hierbei muss sichergestellt werden, dass der neue Code beim Start der App zuerst ausgeführt wird. Dies wird garantiert, wenn Aurasium in der Manifest-Datei als *Application*-Klasse definiert ist.

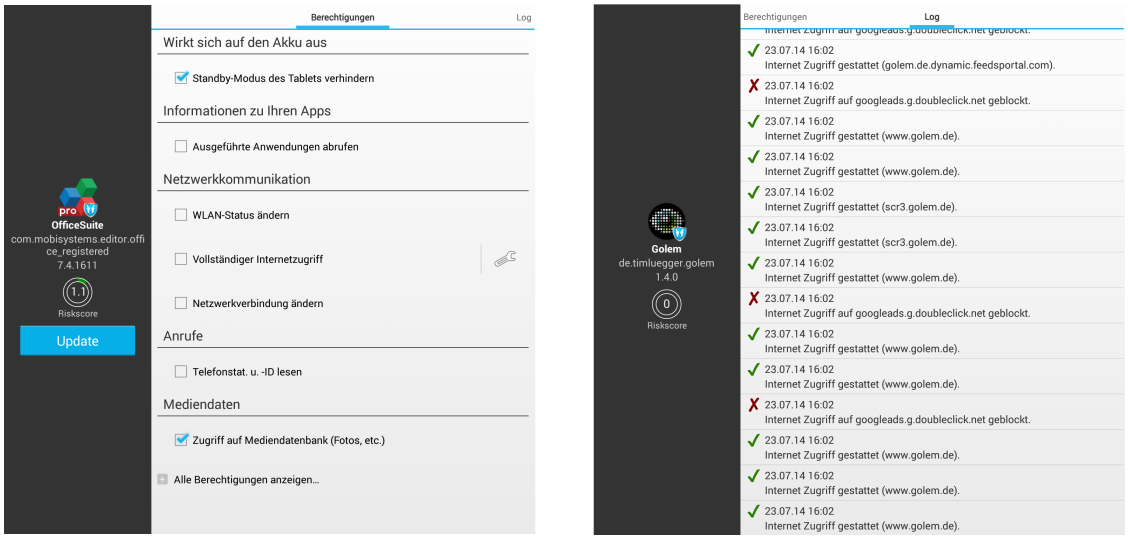
### 3.1.4 AppGuard

*AppGuard* [BGH+13a, BGH+13b] ist ähnlich konzipiert wie Aurasium, arbeitet jedoch auf einer anderen Ebene des Systems. Auch die Wahrnehmung und Steuerung durch den Anwender unterscheidet sich sehr stark. Während Aurasium bei jeder zur Laufzeit auftretenden Berechtigungsanfrage die Entscheidung dem Nutzer überlässt, erlaubt AppGuard, kritische Berechtigungen über eine Permission-Liste der App global zu deaktivieren (siehe Abb. 3.3a).

Implementierungstechnisch werden bei AppGuard - anders als bei Aurasium - Anfragen an das System nicht innerhalb des nativen Codes abgefangen, sondern bereits auf Ebene der Dalvik VM. Hierbei werden die Funktionszeiger auf den Bytecode der tatsächlichen Methoden durch Zeiger auf Methoden des Monitors ersetzt. Dieser überprüft den jeweiligen Zugriff auf Basis der über AppGuard erlaubten Rechte und entscheidet, ob der Aufruf zugelassen wird.

AppGuard reagiert über einen *Broadcast Receiver* automatisch auf die Installation einer neuen App. Auf Wunsch des Anwenders hin kann die App dann mit zusätzlichen Komponenten ergänzt werden, die die Überwachung ermöglichen. Nachdem die neue Version der App installiert wurde, können einzelne Berechtigungen über das AppGuard-Interface deaktiviert werden. Die Liste entspricht dabei der Menge der von der App geforderten Permissions, zusätzlich hat jede App die Berechtigung ”Zugriff auf Mediendatenbank”; diese erfordert in Android kein spezielles Permission-Element, kann aber trotzdem unerwünscht sein und somit blockiert werden.

Manche Berechtigungen, wie z.B. der Internetzugriff, können zusätzlich verfeinert werden. Hierbei erstellt AppGuard eine Liste aller Server, mit denen sich die App seit Neuinstallation verbinden wollte. Der Anwender kann zusätzlich zur globalen Einstellung der Berechtigungsliste einzelne Domains blockieren oder freigeben.




**(a)** Deaktivierung einzelner Berechtigungen

**(b)** Logdatei mit blockierten/erlaubten Zugriffen

**Abbildung 3.3:** AppGuard

Außerdem ist für jede App eine vollständige Logdatei (siehe Abb. 3.3b) aller Zugriffe abrufbar, durch die der Nutzer das Verhalten der App verfolgen kann. Hierbei wird jeweils markiert, welche Zugriffe von AppGuard erlaubt bzw. blockiert wurden. Dadurch können die Auswirkungen von Berechtigungsänderung unmittelbar beobachtet werden.

Durch das Neupacken und die damit entstandene neue Signatur der App sind Updates aus dem AppStore nicht mehr möglich. AppGuard bietet dafür einen eigenen Update-Mechanismus an, der den Anwender über neue Versionen informiert und die Installation übernimmt. Dadurch gehen gespeicherte Daten nicht verloren.

### 3.1.5 Bewertung

Alle Systeme, die eine Manipulation der Apps selbst erfordern, haben den Nachteil, dass beim Ändern des Quellcodes und dem darauffolgenden Neupacken des APKs nicht die ursprüngliche Signatur des Entwicklers verwendet werden kann. Diese dient als Nachweis der Urheberschaft und ist notwendig, um Apps desselben Entwicklers zu erkennen und direkte Updates zu ermöglichen.

Die Beziehung zwischen verschiedenen Apps kann beim Neusignieren erhalten bleiben, indem Apps mit demselben Originalzertifikat dieselbe neue Signatur enthalten. Dies erfordert das Speichern der Originalinformation sowie ein 1:1-Mapping auf neu generierte Signaturen, funktioniert aber nur, wenn alle Apps auf einem Gerät bzw. von derselben Instanz der Software neu gepackt wurden.

Mit diesem Verfahren ist es allerdings nicht mehr möglich, automatische Updates zu beziehen, da die Signatur der App nicht mit der aus dem AppStore übereinstimmt. Um eine neue Version zu installieren, müsste die eigene Version der App zunächst entfernt werden, was aus Anwendersicht unerwünscht sein kann, da etwaige lokal gespeicherte Daten verloren gehen würden. AppGuard umgeht dieses Problem, indem ein eigener Update-Mechanismus angeboten wird, jedoch müssen neue Versionen trotzdem manuell installiert werden.

Ebenfalls problematisch ist die rechtliche Seite, da eine Änderung von Software, auch wenn sie automatisiert geschieht, einen Eingriff in das Urheberrecht des Entwicklers darstellt. Auch ist es damit möglich, Werbung zu blockieren, die für viele freien Apps die Finanzierungsgrundlage darstellen. Google hat beispielsweise AppGuard wieder aus dem PlayStore entfernt, da sie gegen die allgemeinen Geschäftsbedingungen verstößt [Fri12].

Insgesamt können die hier vorgestellten Ansätze direkt verglichen werden (siehe Tabelle 3.1). Dabei fällt schnell auf, dass kein System alle gewünschten Eigenschaften (wie in [SM13] genannt) unterstützt. Laufzeitänderungen sind beispielsweise nur dann möglich, wenn das Kontrollsystem direkt auf dem Anwendergerät installiert ist. Diese Systeme erlauben wiederum keine kontextsensitive (von externen Faktoren wie z.B. Standort und Uhrzeit abhängige) Berechtigungssteuerung. Die Absturzsicherheit der neu gepackten Apps ist in den meisten Fällen gewährleistet, wobei in Einzelfällen durch das Umschreiben Fehler entstehen können. Auch im Bezug auf Feedback gibt es unterschiedliches Verhalten, bei AppGuard werden Funktionen, die widerrufene Berechtigungen erfordern, beispielsweise einfach nicht ausgeführt, ohne den Anwender darauf hinzuweisen. Auf der anderen Seite ist AppGuard das einzige System, welches einen Update-Mechanismus für installierte Apps implementiert.

| | Laufzeit-änderung | Absturz-sicher | Kontext-sensitiv | Dummy-Daten | Anwender-Feedback | Auto-Updates |
|---|---|---|---|---|---|---|
| **Dr. A. & Mr. Hide** | Nein | Ja | Ja | Nein | Nein | Nein |
| **I-ARM** | Nein | Ja | Ja | Ja | Ja | Nein |
| **Aurasium** | Ja | Ja | Nein | Nein | Ja | Nein |
| **AppGuard** | Ja | Ja | Nein | Nein | Nein | Ja |

**Tabelle 3.1:** Vergleich der Android IRM-Systeme

Fazit: Im Bezug auf Anpassbarkeit bietet I-ARM Droid die meisten Möglichkeiten, jedoch sind dazu Programmierkenntnisse und ein tieferes Verständnis der Android-Systemarchitektur notwendig, was es für den Einsatz für ”normale” Anwender unbrauchbar macht.

Von allen hier beschriebenen Ansätzen ist AppGuard am benutzerfreundlichsten. Da es direkt als App auf dem jeweiligen Gerät installiert wird, muss der Anwender nicht erst manuell die APKs herunterladen, diese umwandeln und dann die neue Version auf das Zielgerät spielen. Durch die Möglichkeit, Apps mit einem Klick zu konvertieren und Updates ohne Datenverlust zu installieren, entsteht gegenüber der Installation direkt aus dem AppStore nur ein geringer Mehraufwand.

## 3.2 Erweiterung des Android-Frameworks

Da jeder auf Inline Reference Monitoring (IRM) basierende Ansatz eine Manipulation des Quellcodes der jeweiligen Apps erfordert, kann nicht garantiert werden, dass die Funktionalität der Apps dabei vollständig bestehen bleibt. In Einzelfällen können die Apps nach der Konvertierung auch ohne tatsächliche Einschränkung der Berechtigungen teilweise oder vollständig fehlerhaft sein[2].

Will man die Berechtigungen von Apps ohne Manipulation von deren Quellcode einschränken, ist dies nur auf Systemebene durch eine Änderung am Android-Framework selbst möglich. Zur Installation einer angepassten Version des Android Betriebssystems muss das Zielgerät gerootet sein.

### 3.2.1 Apex

*Apex (Android Permission Extension Framework)* [NKZ10] ist eine angepasste Version des Android-Betriebssystems, die es dem Anwender ermöglicht, direkt bei der Installation einer App einzelne Berechtigungen zu verweigern oder einzuschränken.

Eine Erweiterung des *Package Installers* namens *Poly* erlaubt für jede einzelne Berechtigung die Wahl zwischen drei Möglichkeiten: *allow, deny, constrain*. Neben der im Android-System bisher nicht vorhandenen Möglichkeit, einzelne Berechtigungen überhaupt nicht zu gewähren, ist es damit ebenfalls möglich, den Zugriff auf gewisse Dienste benutzerdefiniert einzuschränken. Ein Beispiel

[2]Bei einem Test von AppGuard mit 30 Apps funktionierten drei davon nach der Konvertierung nicht mehr richtig.

hierfür wäre, die Berechtigung SEND_SMS zu limitieren, so dass nur eine gewisse Anzahl Nachrichten pro Tag erlaubt sind oder diese nur innerhalb bestimmter Uhrzeiten verschickt werden können.

Apex erlaubt außerdem, die Richtlinien von bereits installierten Apps nachträglich zu bearbeiten. Hierzu wurde in den Einstellungen ein Link auf das Poly-Interface für das Richtlinien-Management erstellt. Alle Richtlinien sind ähnlich wie die originalen Android-Berechtigungen in XML-Dateien gespeichert. Diese enthalten unterhalb der UID der jeweiligen App alle Einschränkungen, die durch den Anwender festgelegt wurden.

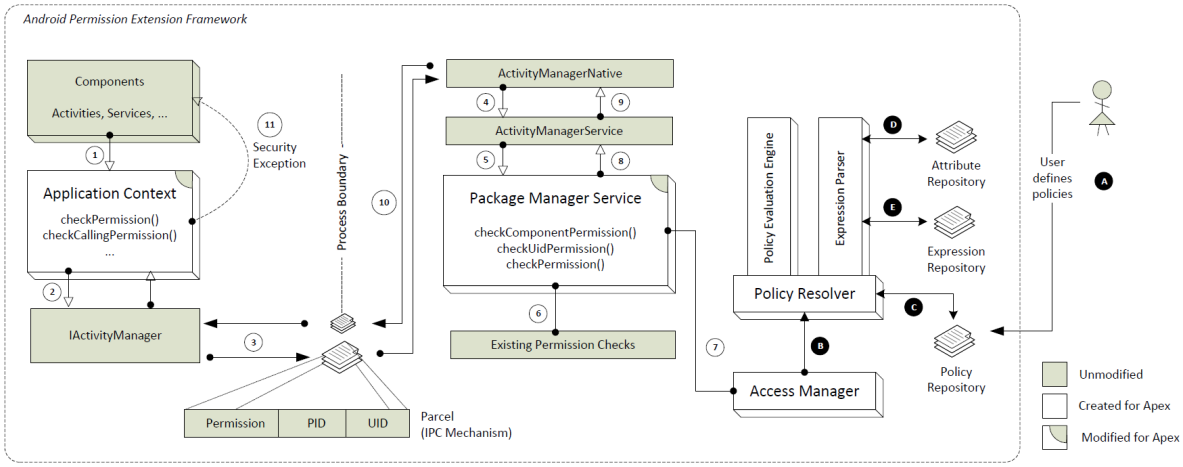



**Abbildung 3.4:** Erweiterung des Android-Frameworks durch Apex [NKZ10]

Wie in Kapitel 2.3 beschrieben, wird jeder Berechtigungscheck über den *Application Context* an den *ActivityManagerService* weitergeleitet. Die darin enthaltene Methode *checkPermission()* ist der einzige öffentliche Eingangspunkt für Berechtigungschecks[3] und dadurch gut dafür geeignet, Anpassungen am Berechtigungssystem vorzunehmen.

Bei Apex wurde der *PackageManagerService* dahingehend erweitert, dass nach den bestehenden Checks die Anfrage an einen neu implementierten *Access Manager* weitergegeben wird (siehe Abb. 3.4). Dieser prüft mithilfe eines *Policy Resolvers* auf Basis der geforderten Permission und der UID der App, ob die Berechtigung gewährt werden soll oder nicht.

Als Teil des erweiterten Berechtigungschecks von Apex wurde für Rückgabewerte eine neue Konstante namens PERMISSION_CONSTRAINT_CHECK_FAILED definiert, für den Fall dass eine Berechtigung aufgrund der vom Anwender definierten Richtlinien verweigert wurde. Diese wird von *enforce()* mit einer besonderen Instanz von *SecurityException* verarbeitet, die den Anwender darüber informiert.

Apex ist somit in der Lage, benutzerdefinierte Einschränkungen aller Berechtigungen jeder App durch den Anwender zu ermöglichen. Ein Nachteil dieser Implementierungsweise ist, dass die Methode *checkPermission()* keine Informationen außer der UID der App selbst enthält, wodurch inhaltsbezogene Richtlinien (z.B. IP-Blacklists bei Internet-Anfragen) nicht möglich sind.

[3] laut Quellcode-Kommentar unter http://goo.gl/rW1th, Zeile 5763f.

### 3.2.2 CRêPE

Während Apex die Möglichkeit bietet, einzelne Berechtigungen von Apps einzuschränken, ist dies nur durch absolute Richtlinien möglich, die vom Anwender definiert werden und ab diesem Zeitpunkt so lange gelten, bis sie manuell angepasst werden. Ein Ansatz, der eine flexiblere kontextabhängige Steuerung von Zugriffsberechtigungen ermöglicht, ist CRêPE [CCFZ12]. Hierbei liegt der Fokus darin, verschiedene *Policies* parallel zu definieren und abhängig von der Umgebung zu entscheiden, welche davon zum jeweiligen Zeitpunkt aktiv sein sollen.

Die so definierten Richtlinien lassen sich in zwei Kategorien einteilen: *Access Control Policies*, die Zugriffsregeln festlegen, und *Obligation Policies*, die Aktionen definieren, die vom System zu bestimmten Zeitpunkten ausgeführt werden sollen (beispielsweise Starten oder Stoppen einer App).

Policies werden in einer Art Matrix zwischen den zugehörigen Subjekten (Apps) und Objekten (Apps, Ressourcen) gespeichert. Der zugeordnete Wert ist eine Regel, die die Anweisung *access* bzw. *deny* sowie eine Priorität enthält. Existieren zu einem bestimmten Zeitpunkt mehrere aktive Regeln, entscheidet die Priorität, welche tatsächlich gilt.

Regeln können abhängig von verschiedenen Kontexten definiert werden. Diese dürfen Informationen enthalten, die über externe Sensoren ermittelt werden können, beispielsweise mit GPS (aktueller Standort), Netzwerkadapter (online oder offline) oder Bluetooth. Hierfür wurde das Betriebssystem durch einen *ContextDetector* erweitert, der Änderungen in der Umgebung registriert und den *ManagerService* entsprechend informiert (siehe Schritte 5-7 in Abb. 3.5).

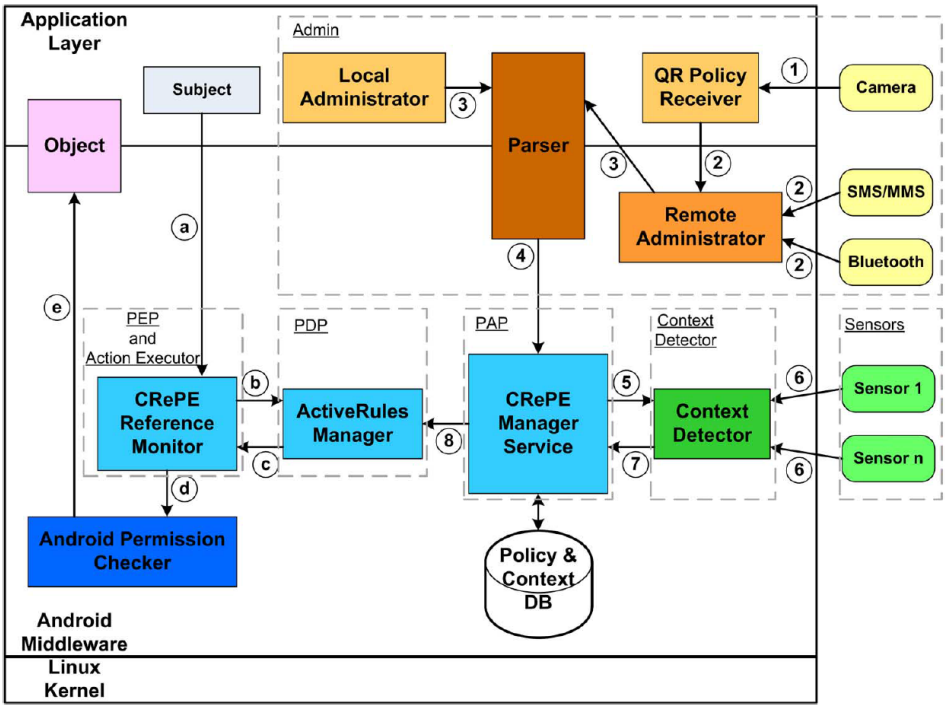



**Abbildung 3.5:** Übersicht der CRêPE-Architektur [CCFZ12]

Der *ActiveRulesManager* erhält die Informationen über den aktuellen Kontext (Schritt 8) und prüft, ob aufgrund einer Änderung des Kontextes bestimmte Aktionen ausgeführt werden müssen (beispielsweise Deaktivierung einer Hardwarekomponente). Wenn ja, wird der *CRêPE Reference Monitor* informiert und die Anweisung durch den darin enthaltenen *ActionExecutor* ausgeführt.

Wie in Kapitel 2.3 beschrieben, wird jeder Permission-Check des Betriebssystems über die Methode *checkPermission()* im ActivityManagerService durchgeführt. Diese Methode wurde hierbei dahingehend erweitert, dass zuerst anhand des Reference-Monitors überprüft wird, ob die aktuell aktiven Policies diesen Zugriff erlauben (Schritte b und c). Ist dies der Fall, wird der normale Berechtigungscheck durchgeführt (Schritt d).

Eine Besonderheit an CRêPE gegenüber anderen vergleichbaren Systemen sind die verschiedenen Möglichkeiten, Policies auf einem Gerät zu installieren. Neben dem Erstellen von eigenen Einträgen durch den Anwender selbst *(Local Administrator)* können diese auch von externen Quellen *(Remote Administrator)*, beispielsweise über SMS, Bluetooth oder das Einlesen eines QR-Codes empfangen werden (siehe *Admin*-Komponente in Abb. 3.5 rechts oben).

Alle installierten Policies müssen ein gültiges Zertifikat enthalten; jeder Administrator hat hierbei eine eindeutige Identität sowie eine maximal erlaubte Priorität, die einzelnen Regeln zugeordnet werden kann. Damit ist es möglich, dass Richtlinien des Remote-Administrators (beispielsweise eine Firma) die Regeln des Anwenders (ein Angestellter dieser Firma) überschreiben kann, da dieser eine höhere Priorität vergeben darf als der Anwender selbst.

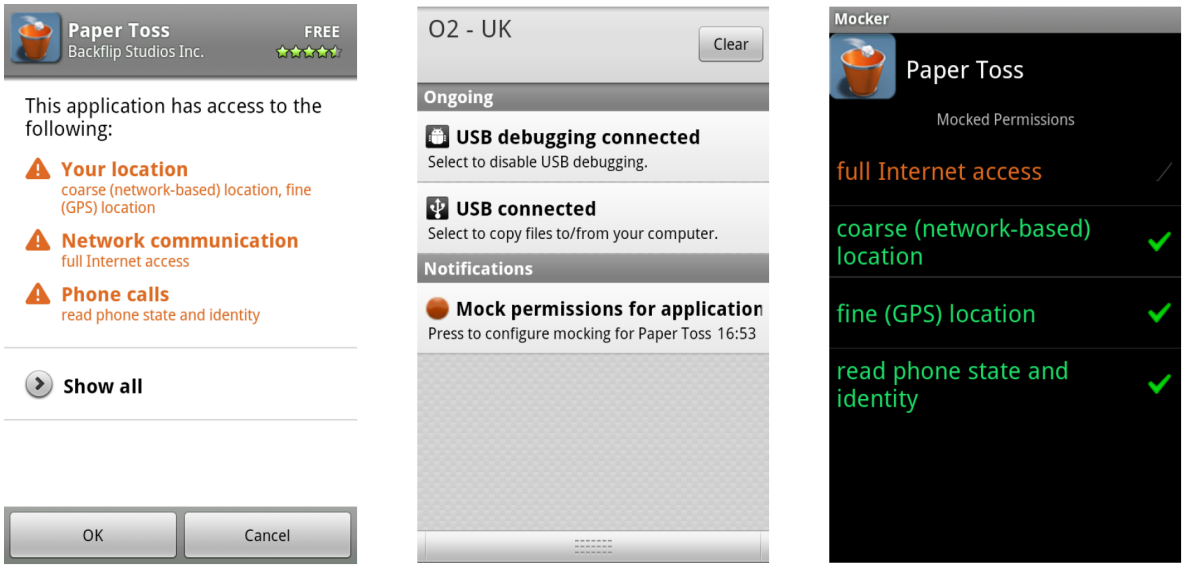
Insgesamt liegt der Fokus dieses Ansatzes auf der kontextabhängigen Steuerung von Berechtigungen, die sicherstellen kann, dass zu bestimmten Zeitpunkten und in bestimmten Situationen einzelne Funktionen (die generell erlaubt sind) unterbunden werden können, ohne dass der Anwender manuell Änderungen an Berechtigungsregeln vornehmen muss. Auch können globale Richtlinien ”von oben” durchgesetzt werden, die der Anwender selbst nicht manipulieren kann.

### 3.2.3 MockDroid

Viele Apps benötigen gewisse Berechtigungen nur für zusätzliche optionale Features. Auch wenn ein Anwender ein Feature überhaupt nicht verwenden möchte, muss er trotzdem die Berechtigung erteilen, da die App sonst nicht installiert werden kann. Um private Daten wie z.B. Standort und Kontakte vor möglichem Missbrauch zu schützen, besteht ein Ansatz darin, dass das System Zugriffe auf solche Daten abfängt und stattdessen Dummy-Daten zurückgibt.

Das System *MockDroid* [BRSS11] erlaubt Dummy-Daten für Standort, SMS, Kalender, Kontakte und Geräte-ID. Außerdem können Internet-Anfragen so abgefangen werden, dass ein Timeout auftritt und die App dadurch den Eindruck hat, es bestehe keine Verbindung. Auch das Senden und Empfangen von Broadcasts kann verhindert werden.

Bei Installation einer neuen App werden zunächst alle Rechte zugeteilt. Der Anwender wird hierbei durch das Android-System über die benötigten Berechtigungen informiert und muss diese wie gewohnt bestätigen. Daraufhin erscheint in der Benachrichtigungsleiste ein Hinweis, über den man zu einer Manager-Oberfläche gelangt, wo man einzelne Berechtigungen deaktivieren und damit durch Dummy-Daten ersetzen kann (siehe Abb. 3.6).




**(a)** Information über benötigte Berechtigungen (Standard)

**(b)** Benachrichtigung nach Installation einer App

**(c)** Benutzerdefinierte Anpassung der Berechtigungen

**Abbildung 3.6:** MockDroid [BRSS11]

Um Dummy-Daten zu speichern, wurde dem System eine zusätzliche Gruppe namens *mock* hinzugefügt, die diese innerhalb eines festgelegten Verzeichnisses für alle Apps verwaltet. Außerdem ist für den PackageManager ein Service implementiert, der bei Änderungen an diesen Daten ein Update der Berechtigungsinformation im Hauptspeicher veranlasst.

Um Änderungen an den Daten vorzunehmen, wurde die App *Mocker* entwickelt, die durch eine neu definierte Systemberechtigung als einzige App auf das entsprechende Verzeichnis zugreifen darf. Damit ist es möglich, im laufenden Betrieb die Einstellungen für jede installierte App zu ändern.

Will eine App auf geschützte Daten zugreifen, wird zunächst der Berechtigungscheck des Application Frameworks aufgerufen. Die entsprechende Methode wurden hierbei so modifiziert, dass nach der bereits existierenden Prüfung außerdem noch kontrolliert wird, ob die jeweilige Ressource für die abfragende App durch MockDroid freigegeben oder blockiert werden soll. In letzterem Fall werden ggf. passende Dummy-Daten zurückgegeben, z.B. eine leere Datenbank.

Dieses Verfahren hat den Vorteil, dass die App trotz effektiv fehlender Zugriffsberechtigung normal arbeiten kann und keine Fehler vom System zurückbekommt. Die App selbst ”merkt” dabei nicht, dass die verwendeten Daten nicht echt sind, da das Datenformat den Erwartungen entspricht.

Anwender sind somit in der Lage, selbst festzulegen, welche Daten sie jeder App zur Verfügung stellen möchten, und können im laufenden Betrieb testen, welche Auswirkungen dies jeweils auf deren Funktionalität hat.

### 3.2.4 AppFence

Die zuvor beschriebenen Verfahren zeigen einerseits, wie Berechtigungen auf Systemebene widerrufen werden können (mit der Gefahr, dass die App als Konsequenz möglicherweise abstürzt) und andererseits, wie durch Rückgabe von Dummy-Daten die Kernfunktionalität bewahrt werden kann, ohne dass Fehler auftreten.

Ein Ansatz, der diese beiden Methoden miteinander kombiniert, ist *AppFence* [HHJ+11]. Hierbei wurde zunächst das Monitoring-Tool *TaintDroid* [EGC+10] verwendet, um bestimmte geschützte Daten markieren und deren Weitergabe verfolgen zu können. Darauf aufbauend wurden zwei verschiedene Verfahren implementiert, um die Weitergabe dieser Daten an externe Server zu unterbinden:

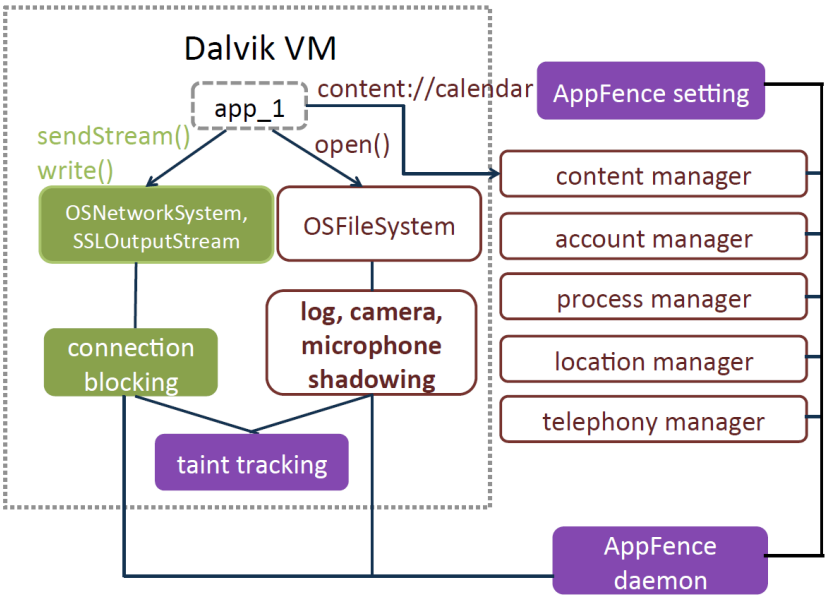



**Abbildung 3.7:** AppFence: Anpassung der Framework-Dateien, um Rückgabe von manipulierten Daten zu ermöglichen (dunkelrot), Blockade ausgehender Nachrichten (grün) [HHJ+11]

*Data Shadowing* - Ähnlich wie auch bei MockDroid werden hierbei Daten weitergegeben, deren Format und Inhalt einem vom System erwarteten Datensatz entspricht. In vielen Fällen ist dies einfach durch eine leere Datenbank zu bewerkstelligen (z.B. bei Kalender, Kontakten, Nachrichten). Manchmal ist es aber notwendig, tatsächliche (scheinbar korrekte) Informationen zurückzugeben, da die App diese überprüft oder zur Ausführung zwingend benötigt. Ein Beispiel hierfür wäre eine (durch AppFence gefälschte) IMEI-Nummer des Geräts oder auch per Default definierte GPS-Koordinaten.

Im Gegensatz zu anderen Systemen, die Zugriffe durch Widerruf von Berechtigungen durch Änderungen am Application-Framework unterbinden, werden hierbei direkt die Java-Methoden manipuliert, die die tatsächlichen Daten zurückgeben (siehe Abb. 3.7). Dabei handelt es beispielsweise um die Manager-Klassen des Application-Frameworks (z.B. für Standort, Telefoninformationen und sämtliche Content-Provider), andere Daten (wie Kamera und Mikrofon) werden über das Dateisystem abgerufen.

*Exfiltration blocking* - Hierbei geht es darum, die Weitergabe von Daten über das Internet nur in bestimmten Fällen zu unterbinden. Dies ist sinnvoll, wenn die Übertragung privater Informationen für die Hauptfunktionalität einer App notwendig ist (beispielsweise der korrekte Standort bei einem Navigationssystem), dieselben Informationen aber nicht an den Server eines Werbetreibers gesendet werden sollen.

Dabei gibt es in Bezug auf die Kommunikation mit der entsprechenden App wiederum zwei verschiedene Ansätze: Entweder wird die entsprechende Nachricht durch das System einfach nicht weitergegeben, wobei die App darüber nicht informiert wird, oder das System simuliert den Offline-Modus und gibt eine entsprechende Meldung an die App zurück.

Durch verschiedene Tests wurde gezeigt, dass mit einem dieser Verfahren alleine nicht garantiert werden kann, dass ein Großteil der Apps weiterhin fehlerfrei läuft. Insgesamt ist eine Kombination der beiden Ansätze am erfolgversprechendsten, wobei vom jeweiligen Anwender allerdings ein gewisses Grundverständnis vorausgesetzt wird, um diesbezüglich informierte Entscheidungen treffen zu können. Um die Verwendung des Systems zu erleichtern, wäre es beispielsweise denkbar, eine zentrale Anlaufstelle zu eröffnen, bei der durch eine große Anzahl von Nutzern *(crowdsourcing)* für bekannte Apps die bestmöglichen Einstellungen ermittelt werden können.

### 3.2.5 Bewertung

Ein Nachteil der hier beschriebenen Methoden ist, das eine Anpassung des Android-Systems eine große Hürde darstellt. Um ein derartiges System überhaupt installieren zu können, muss das entsprechende Gerät gerootet sein, wodurch etwaige Garantieansprüche verfallen. Ein weiteres Problem ist, dass bereits jetzt jedes Geräte-Modell eine durch den Hersteller speziell angepasste Version des Android-Betriebssystems enthält. Die dadurch entstandene Fragmentierung der existierenden Android-Versionen würde sich durch die Verbreitung der hier beschriebenen Systeme nur noch weiter verschlimmern.

Abgesehen von dieser Problematik sind die hier beschrieben Ansätze aus Anwendersicht sehr komfortabel, da sie sich im laufenden Betrieb genauso verhalten wie bekannte Android-Systeme und lediglich um die Möglichkeit ergänzt wurden, einzelne Berechtigungen zu widerrufen.

Dadurch, dass die Apps selbst nicht verändert werden und damit die Originalsignatur erhalten bleibt, können diese normal aus den jeweiligen AppStores installiert und upgedatet werden. Auch besteht keine Gefahr, dass eine App durch Integration eines Monitors fehlerhaft wird.

Jeder Anwender kann selbst Richtlinien definieren, die festlegen, auf welche Daten die jeweilige App zugreifen darf. Dadurch, dass Richtlinien jederzeit geändert werden können, sind Anwender dazu in der Lage, genau zu beobachten, welche Auswirkungen diese auf die jeweilige App haben.

Die vorgestellten Systeme können analog zu Kapitel 3.1.5 auf Basis ihrer Eigenschaften verglichen werden. Auch hier sieht man schnell, dass es kein System gibt, was alle gewünschten Möglichkeiten bietet. Da hier sämtliche Berechtigungen über das Betriebssystem selbst gesteuert werden, können diese im laufenden Betrieb beliebig angepasst werden.

Alle weiteren Eigenschaften werden aber nur von manchen Systemen unterstützt (siehe Tabelle 3.2). Bei Systemen, die Zugriffe auf Systemebene blockieren, kann bei bestimmten Implementierungsansätzen insbesondere das Problem bestehen, dass Apps aufgrund fehlender Rechte abstürzen. Das liegt am Ablauf eines Berechtigungschecks (siehe Kapitel 2.3), der, falls er fehlschlägt, den Programmablauf durch eine SecurityException unterbricht, die in Apps meist nicht abgefangen wird. Wird das System an einer anderen Stelle erweitert, besteht dieses Problem nicht.

| | Laufzeit-änderung | Absturz-sicher | Kontext-sensitiv | Dummy-Daten | Anwender-Feedback | Remote Admin |
|---|---|---|---|---|---|---|
| **Apex** | Ja | Nein | Ja | Nein | Ja | Nein |
| **CRêPE** | Ja | Nein | Ja | Nein | ? | Ja |
| **MockDroid** | Ja | Ja | Nein | Ja | Nein | Nein |
| **AppFence** | Ja | Ja | Nein | Ja | Nein | Nein |

**Tabelle 3.2:** Vergleich der erweiterten Android-Systeme

Generell setzen die hier beschriebenen Verfahren verschiedene Schwerpunkte. Während es bei Apex und CRêPE primär um die (kontextbezogene) Restriktion von Zugriffen auf bestimmte Informationen geht, sind AppFence und MockDroid speziell darauf bedacht, Dummy-Daten zu verwenden, um die Funktionalität einer App zu bewahren, ohne tatsächlich private Daten preisgeben zu müssen.

Letztendlich ist es abhängig vom gewünschten Anwendungszweck, welches System tatsächlich sinnvoll ist. Für private Anwender reicht es häufig, wie mit Apex und MockDroid einfach nur selbst Berechtigungen widerrufen zu können - wenn eine App dadurch nicht mehr funktioniert, können die Richtlinien schnell angepasst werden. Komplexere Systeme wie CRêPE sind dagegen eher auf Firmenebene gut geeignet, um globale Richtlinien durchsetzen zu können, die der Anwender selbst nicht umgehen kann.

Insgesamt ist aber deutlich zu sehen, dass ein angepasstes Betriebssystem zur Berechtigungssteuerung mehr Vorteile bringt als die Manipulation jeder einzelnen App über einen Konverter. Insbesondere aus Sicht von Anwendern ist letzteres doch eher umständlich, da bei der Installation jeder einzelnen App aktiv Schritte durchgeführt werden müssen, um deren Berechtigungen widerrufbar zu machen. Ein angepasstes Betriebssystem wird dagegen einmalig auf einem Gerät installiert und ermöglicht dies ohne weiteres Zutun des Anwenders.

Allerdings erfüllt keine der hier vorgestellten Ergänzungen des Android-Betriebssystems alle Anforderungen, die an eine flexible Berechtigungssteuerung gestellt werden. Das im folgenden Kapitel vorgestellte Konzept beschreibt einen Ansatz, der die Vorteile aller bisherigen Systeme miteinander kombiniert.

# 4 Die Privacy Management Platform

Die *Privacy Management Platform (PMP)* [Sta13a] ist ein alternatives Berechtigungssystem für Android, das eine kontextbezogene Steuerung von App-Berechtigungen durch den Anwender ermöglicht.

Der Grundgedanke des Systems besteht darin, Rechte nicht einer gesamten App zuzuweisen, sondern nur den Komponenten, die diese auch tatsächlich benötigten. Dadurch führt der Widerruf einzelner Berechtigungen lediglich zur Deaktivierung der entsprechenden Funktionen.

Im Gegensatz zu anderen alternativen Systemen baut die PMP hierbei nicht auf dem klassischen Permission-Modell von Android auf, sondern führt eine alternative Form des Berechtigungsmanagements ein. Dies soll garantieren, dass Apps nur über die PMP Zugriff auf private Daten erhalten können, um dem Anwender vollständige Kontrolle darüber zu geben, welche App auf welche Informationen zugreifen darf.

## 4.1 Aufbau und Grundbegriffe

Abbildung 4.1 zeigt eine Übersicht der verschiedenen Bestandteile des PMP-Berechtigungssystems in einer an UML angelehnten Notation.Die einzelnen Komponenten und deren Zusammenhänge werden im Folgenden erläutert.

### 4.1.1 Ressourcen und Privacy Settings

PMP-Apps verwenden keine klassischen Android-Permissions und haben deshalb auch keinen direkten Zugriff auf geschützte Informationen über das Application-Framework. Die Interfaces für benötigte Daten werden von einzelnen *Ressourcen* bereitgestellt. Hierbei werden drei verschiedene Modi unterstützt: *Rückgabe korrekter Daten*, *Rückgabe veränderter Daten* und *Rückgabe zufälliger Daten*. Eine Menge von Ressourcen kann zu einer *Ressourcengruppe* zusammengefasst werden.

Wie die tatsächlichen Daten bezogen werden, ist dem Entwickler der Ressource freigestellt. Hierbei kann die entsprechende Hardware (z.B. GPS) direkt angesprochen werden, ebenfalls ist aber möglich, einen allgemeineren Ansatz zu verwenden und den Standort auf Basis der jeweils auf dem Gerät verfügbaren Informationen zu ermitteln (z.B. auch über den WLAN-Adapter oder den Netzbetreiber). Ebenfalls ist es möglich, die Verwendung einer Ressource auf bestimmte Funktionen (Service Features) oder sogar nur auf einzelne Apps einzuschränken.

Jede Ressource definiert verschiedene *Privacy Settings*, mit denen der Anwender steuern kann, welche Zugriffe dem jeweilige Feature erlaubt werden sollen. Im einfachsten Fall sind die Werte *true* oder *false* vorhanden, um den Zugriff entweder zu erlauben oder nicht. Manchmal kann es aber auch

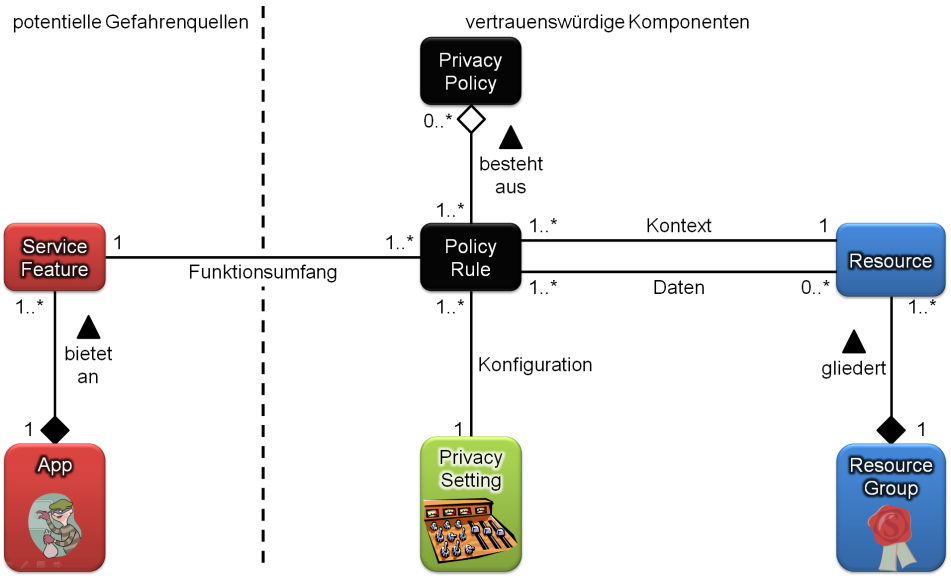



**Abbildung 4.1:** Komponenten der PMP [Sta13b]

sinnvoll sein, mehrere Abstufungen zu ermöglichen. Bei einer Ressource zur Standortbestimmung wäre es beispielsweise denkbar, den Anwender zusätzlich entscheiden zu lassen, ob *CoarseLocation* oder *FineLocation* verwendet werden soll.

Ebenfalls sind über die Privacy Settings zusätzliche kontextuelle Einschränkungen definierbar. Diese können einerseits von externen Faktoren wie Standort und Uhrzeit abhängen, andererseits aber auch auf Basis des jeweiligen Zugriffs (beispielsweise abhängig von der Adresse einer Internet-Anfrage) getroffen werden.

Ressourcen sind nicht direkt in die PMP eingebaut, sondern können im laufenden Betrieb bei Bedarf hinzugefügt werden[1]. Benötigt eine neue App eine noch fehlende Ressource, wird diese automatisch aus einem geschützten Datenarchiv nachinstalliert. Dies hat den Vorteil, dass Ressourcen unabhängig von der PMP selbst aktualisiert oder auch neu veröffentlicht werden können, wodurch das System frei erweiterbar ist.

### 4.1.2 Service Features

Alle Teilfunktionen einer App, die besondere Berechtigungen erfordern, werden auf einzelne *Service Features* abgebildet. Hierbei enthält jedes Service Feature neben einem Namen und einer Beschreibung die Menge der dafür benötigten Ressourcen und Privacy Settings. Jedes einzelne Service Feature kann im laufenden Betrieb entweder aktiviert oder deaktiviert sein, wodurch die App auf die entsprechenden Funktionen und Berechtigungen eingeschränkt ist.

[1]Im PMP-Protoyp sind Ressourcen als eigene Apps implementiert, die Schnittstellen zur PMP enthalten.

Dieses Modell stellt die größte Besonderheit der PMP im Vergleich mit anderen Berechtigungssystemen dar, da es ein völlig neues Konzept implementiert. Während es bei anderen Systemen passieren kann, dass Apps aufgrund fehlender Berechtigungen abstürzen, löst die PMP dieses Problem, indem die entsprechenden Funktionen bei der Ausführung der App übersprungen werden. Der Anwender kann dadurch direkt verfolgen, welche Auswirkung das Aktivieren und Deaktivieren bestimmter Features auf den Funktionsumfang der App hat.

### 4.1.3 Privacy Rules und Presets

Um ein Service Feature einer App zu aktivieren, müssen entsprechende Regeln *(Privacy Rules)* definiert sein. Diese bestehen aus dem jeweiligen Feature, der zugehörigen Ressource und dem vom Anwender für diesen Zugriff definierten Privacy Setting. Sind für eine bestimmte Aktion noch keine Regeln vorhanden, wird der Anwender darüber entsprechend informiert. Die Menge aller definierten Regeln bilden die *Privacy Policy*.

Eine beliebige Teilmenge von Regeln kann in ein so genanntes *Preset* exportiert werden. Ein solches Preset kann an andere weitergegeben und wiederum in die PMP importiert werden. Dadurch ist es möglich, unerfahrenen Anwendern die Konfiguration der PMP zu erleichtern, indem der Allgemeinheit fertige Presets aus vertrauenswürdigen Quellen zur Verfügung gestellt werden. Existieren mehrere Presets für eine App, werden alle miteinander kompatiblen Regeln aktiviert; bei Konflikten muss der Anwender entscheiden, welche Regel Priorität hat.

Unter anderem könnten auch Firmen diese Möglichkeit nutzen, um entsprechende Presets für ihre Mitarbeiter zu erstellen. In der heutigen Zeit ist es nicht ungewöhnlich, eigentlich private mobile Geräte auch im Geschäftsleben zu nutzen *(bring-your-own-device)* und damit auf Firmendaten zuzugreifen, was ein nicht unwesentliches Sicherheitsrisiko darstellt. [Pan13] beschreibt, wie das bestehende Preset-Management der PMP erweitert werden kann, um Firmen zu ermöglichen, eigene Richtlinien durchzusetzen, die vom Anwender nicht umgangen werden können.

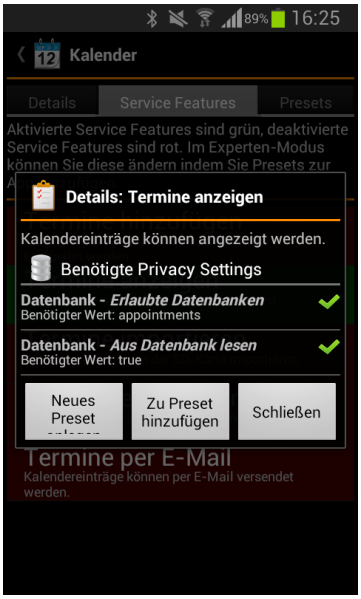
## 4.2 Management

Damit für die PMP entwickelte Apps installiert werden können, muss die PMP selbst bereits auf dem Gerät vorhanden sein. Nach der Installation einer App wird diese automatisch gestartet und registriert sich bei der PMP. Da neuen PMP-Apps standardmäßig keine Berechtigungen gewährt sind, wird nun die Liste der verfügbaren Service Features angezeigt, aus denen der Anwender die gewünschten Funktionen auswählen kann. Danach ist die Installation abgeschlossen und die App kann verwendet werden.

Über das Hauptfenster der PMP (siehe Abb. 4.2a) kann der Anwender auf alle registrierten Apps zugreifen. Dort kann laufend angepasst werden, welche Service Features einer App aktiviert sein sollen (siehe Abb. 4.2b). Unter ”Ressourcen” werden alle auf dem Gerät installierten und sonstige zur Installation bereitstehenden Ressourcen angezeigt.


**(a)** Hauptfenster der PMP


**(b)** Service Features




**(c)** Privacy Rule

**Abbildung 4.2:** PMP-Benutzeroberfläche am Beispiel einer Kalender-App

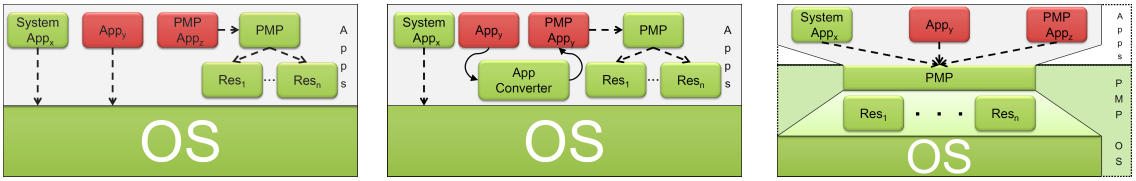
Die PMP kann in zwei verschiedenen Modi betrieben werden, dem *einfachen Modus* und dem *Expertenmodus*. Der einfache Modus umfasst nur die wichtigsten Grundfunktionen der PMP und richtet sich an eher unerfahrene Anwender. Hierbei ist es lediglich möglich, einzelne Service Features ein- und auszuschalten.

Im Expertenmodus können Regeln verschiedenen Presets zugeordnet werden (siehe Abb. 4.2c). Deren Liste ist über das Hauptfenster abrufbar, dort können neue Presets angelegt sowie importiert und exportiert werden. Im Detailfenster eines Presets sind die darin enthaltenen Apps sowie alle zugehörigen Privacy Settings aufgelistet. Diese können durch Änderung der Werte und das Hinzufügen von benutzerdefinierten Kontexten weiter angepasst werden.

## 4.3 Integrationsstrategien

Es gibt prinzipiell drei verschiedene Ansätze, die PMP in ein bestehendes Android-System zu integrieren (siehe Abb. 4.3). Je nach gewählter Strategie werden unterschiedliche Anforderungen an System und Anwender gestellt, gleichzeitig kann nicht bei allen Ansätzen ein vollkommener Schutz privater Daten und die Stabilität aller verwendeten Apps garantiert werden.

Ebenfalls hat die Wahl der Integrationsstrategie großen Einfluss auf die Implementierungsform und damit schlussendlich die Funktionsweise der in Kapitel 5 vorgestellten Gatekeeper-Komponente.




**(a)** Application-Level **(b)** App-Konverter **(c)** Teil des Frameworks

**Abbildung 4.3:** Implementierungsstrategien der PMP [SM14]

### 4.3.1 Application-Level

Die einfachste Form der PMP ist die Implementierung in Form einer App (siehe Abb. 4.3a). Alle Komponenten, inklusive Ressourcen, liegen auf dem Application-Level und benötigen keine besonderen Zugriffe auf das Betriebssystem. Dies hat den Vorteil, dass das System auf jedem Gerät normal installiert werden kann.

Für die PMP entwickelte Apps können somit nach Bedarf eingeschränkt verwendet werden, während klassische Apps nicht beeinflusst werden und somit weiterhin voll funktionsfähig sind.

Dies ist aber gleichzeitig der große Nachteil dieser Implementierungsart, da diese Apps weiterhin uneingeschränkt auf private Daten zugreifen dürfen. Ebenfalls könnten Apps, die scheinbar mit der PMP zusammenarbeiten, durch die zusätzliche Verwendung klassischer Permissions entgegen dem Wunsch des Anwenders trotzdem direkten Zugriff auf das Application-Framework erlangen.

Insgesamt ist dieser Ansatz also zwar zu Testzwecken ausreichend[2], kann aber keine tatsächliche Sicherheit der privaten Daten garantieren, was ihn für die Praxis untauglich macht.

### 4.3.2 App-Konverter

In Kapitel 3.1 wurde beschrieben, wie alternative Berechtigungssysteme durch den Einsatz von Konvertern bestehende Apps anpassen, so dass eine Steuerung der Berechtigungen dieser Apps ohne Eingriff in das Betriebssystem möglich wird.

Bei der PMP könnte ein derartiger Ansatz so aussehen, dass alle klassischen Permissions aus dem Manifest der App entfernt werden, so dass die App nicht mehr direkt auf das Application-Framework zugreifen kann. An deren Stelle könnten Schnittstellen auf die entsprechenden PMP-Ressourcen in den Quellcode der App integriert und der Bytecode entsprechend umgeschrieben werden, so dass die App effektiv in eine PMP-App ”umgebaut” wird (siehe Abb. 4.4).

Dies hätte den Vorteil, dass die PMP trotzdem auf Anwendungsebene implementiert werden könnte, während klassische Apps automatisch so angepasst werden, dass sie nur noch über die PMP auf private Daten zugreifen können.

[2]Der Protoyp der PMP ist auf Application-Level implementiert: https://code.google.com/p/pmp-android/

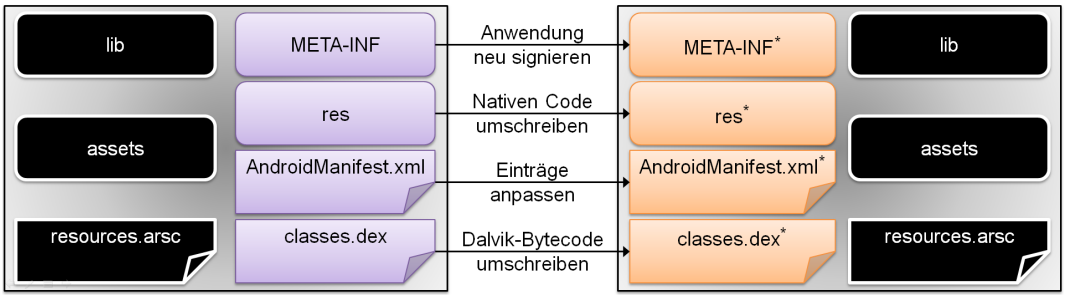



**Abbildung 4.4:** PMP App-Konverter [Sta13b]

Während dadurch die Sicherheit der Daten weitgehend gewährleistet ist, kann nicht garantiert werden, das dieser automatische Prozess bei allen Apps erfolgreich verläuft. Erfahrungen mit ähnlichen Systemen (z.B. AppGuard) haben gezeigt, dass vereinzelt Apps nach dem Konvertieren nicht mehr richtig funktionieren. Ein weiteres Problem ist die rechtliche Seite, da eine Manipulation des Bytecodes einer App das Urheberrecht des Entwicklers verletzt. Schließlich besteht auch noch das Problem, dass eine durch den Konvertierungsprozess neu signierte App nicht durch automatische Updates aktualisiert werden kann.

### 4.3.3 Anpassung des Application-Frameworks

Die einzige Möglichkeit, ohne Manipulation der Apps selbst Zugriffe auf private Daten zu verhindern, besteht darin, die PMP direkt in das Android-Betriebssystem zu integrieren (siehe Abb. 4.3c). Dadurch können alle Anfragen auf Basis klassischer Android-Permissions vom Application-Framework blockiert werden, während über die PMP angeforderte Ressourcen weiterhin frei zugänglich sind[3].

Dies führt im Umkehrschluss natürlich dazu, dass Apps, die nicht speziell für die Verwendung mit der PMP entwickelt wurden, alle Rechte verlieren und somit größtenteils unbenutzbar werden. Da diese Situation von Seiten des Entwickler auch nicht berücksichtigt wird, ist es sehr wahrscheinlich, dass betroffene Apps einfach abstürzen, sobald eine Methode aufgerufen wird, die besondere Berechtigungen erfordert (siehe Kapitel 2.2).

Während dieses Vorgehen zwar die Sicherheit der privaten Daten garantiert, ist es in der Praxis nicht anwendbar, da nicht davon ausgegangen werden kann, dass es von allen Apps PMP-kompatible Versionen geben wird. Um einen realistischen Kompromiss zwischen Sicherheit und Funktionalität zu treffen, muss der Anwender selbst dazu in der Lage sein, diesen Apps (ähnlich wie bei PMP-Apps) einzelne Berechtigungen gewähren zu können.

Zu diesem Zweck muss der PMP eine neue Komponente hinzugefügt werden, die es zulässt, dass einzelne Zugriffe auf Basis klassischer Berechtigungen vom Application-Framework nicht blockiert werden. Die Konzeption und prototypische Implementierung dieses ”Gatekeepers” wird im folgenden Kapitel beschrieben.

[3]Philipp Scholz hat diese Implementierungsstrategie im Rahmen seiner Diplomarbeit realisiert [Sch13]

# 5 Der PMP-Gatekeeper

Der *Gatekeeper* ist eine im Rahmen dieser Arbeit neu entwickelte PMP-Komponente, die festlegen soll, welche Zugriffe von Legacy-Apps auf private Daten erlaubt bzw. blockiert werden sollen. Die Bezeichnung *Legacy-App* beschreibt hierbei Apps, die nicht für die Verwendung mit der PMP entwickelt wurden und demnach Verweise auf klassische Android-Permissions enthalten.

Für Legacy-Apps sollen Zugriffe auf Basis von definierten Richtlinien entschieden werden, wobei System-Apps aus Stabilitätsgründen uneingeschränkten Zugriff auf das Framework erhalten müssen. Eine der Hauptaufgaben des Gatekeepers ist also, eine App zuverlässig als ”Legacy” oder ”PMP” klassifizieren zu können.

## 5.1 Identifizierung von Apps

Jede App, die für die Verwendung mit der PMP entwickelt wurde, enthält Komponenten, an denen sich dies eindeutig feststellen lässt. Da die APKs aller installierten Apps unter Android in einem öffentlichen Verzeichnis abgelegt sind, können die darin enthaltenen Informationen von anderen Apps ausgelesen werden. Im Folgenden wird beschrieben, wie der Gatekeeper diese Informationen nutzen kann, um Apps zu klassifizieren.

Um die Metadaten einer beliebigen installierten App zu erhalten, kann die Methode *PackageManager.getPackageInfo()* verwendet werden. Diese gibt ein *PackageInfo*-Objekt zurück, aus dem alle im Manifest hinterlegten Informationen ausgelesen werden können.

### 5.1.1 Registrierung bei der PMP

Da sich jede PMP-App beim ersten Start bei der PMP registrieren muss, enthält das Manifest unterhalb des <application>-Elements einen Aufruf der PMP-*RegistrationActivity* (siehe Lst. 5.1). Die Activities einer App können mit *PackageInfo.activities* ausgelesen werden, der Name der zugehörigen Klasse ist dabei unter *ActivityInfo.name* gespeichert.

Hierbei reicht es allerdings nicht, lediglich zu prüfen, ob die RegistrationActivity überhaupt referenziert wird. Um die PMP zu umgehen, könnte sie einfach als Activity eingetragen werden, auch wenn sie nie aufgerufen wird, was zu einer falsch positiven Identifizierung führen würde. Es ist daher ebenfalls notwendig zu überprüfen, ob die entsprechende Activity auch tatsächlich der Launcher der jeweiligen App ist.

**Listing 5.1** Registrierungsaufruf von PMP-Apps im Manifest

```
<activity
      android:name="de.unistuttgart.ipvs.pmp.api.gui.registration.RegistrationActivity"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoTitleBar" >
      <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
      ...
</activity>
```

Dies ist allerdings nicht über das PackageInfo-Element möglich, da es sich bei der Aktion LAUNCHER um einen Intent handelt und diese vom System an einer anderen Stelle gespeichert werden. Die einzige Möglichkeit, diese abzufragen, besteht drin, vom PackageManager eine Liste aller Intents der Launcher-Kategorie anzufordern und daraus wiederum die Namen der passenden Apps zu ermitteln.

### 5.1.2 Referenz des PMP-AppService

Eine weitere Möglichkeit, eine PMP-App anhand ihres Manifests zu identifizieren, besteht über ihre Implementierung der PMP-Service-Schnittstelle. Alle Apps, die die PMP verwenden, um auf Ressourcen zuzugreifen, müssen unterhalb von <application> eine Referenz auf den PMP-AppService enthalten. Darunter befindet sich eine Intent-Liste, die eine Referenz auf den eigenen Paketnamen enthält (siehe Lst. 5.2).

**Listing 5.2** Serviceaufruf von PMP-Apps im Manifest

```
<!-- The service of the app where pmp connects to -->
<service
      android:name="de.unistuttgart.ipvs.pmp.service.app.AppService"
      android:exported="true" >
      <intent-filter>
            <action android:name="de.unistuttgart.ipvs.pmp.apps.bluetoothtestapp">
            </action>
      </intent-filter>
</service>
```

Auch diese Informationen können über vorhandene Java-Methoden ermittelt werden. Mit *PackageInfo.services* erhält man ein Array von *ServiceInfo*-Objekten, die mit der Menge der <service>-Tags in der Manifest-Datei korrespondieren. Mit *ServiceInfo.name* kann wiederum die referenzierte Klasse ermittelt werden.

Allerdings ist es mit dieser Methode leicht möglich, den Gatekeeper zu täuschen, da das Vorhandensein eines solchen Serviceeintrags alleine natürlich nicht garantiert, dass es sich tatsächlich um eine PMP-App handelt.

### 5.1.3 AppInformationSet

Anstatt das Manifest auszulesen, können PMP-Apps auch über ihre Android-Ressourcen identifiziert werden. Jede PMP-App enthält unter *assets/ais.xml* (siehe Lst. 5.3) in ihrem *appInformationSet* eine Liste vorhandener Service Features. Diese Liste wird von der PMP verwendet, um einzelne Funktionen der App anzuzeigen und zu steuern.

**Listing 5.3** Application Information Set einer PMP-App (Auszug)

```
<appInformationSet>
      <appInformation>
            <name>Calendar</name>
            <description>This App is used to test the privacy management platform. It provides
                simple calendar functionalities.</description>
      </appInformation>
      <serviceFeatures>
            <serviceFeature identifier="read">
                  <name>Show entries</name>
                  <description>Calendar entries can be displayed.</description>
                  <requiredResourceGroup
                       identifier="de.unistuttgart.ipvs.pmp.resourcegroups.database"
                       minRevision="1">
                        <requiredPrivacySetting identifier="read">true</requiredPrivacySetting>
                        <requiredPrivacySetting
                             identifier="allowedDatabases">appointments</requiredPrivacySetting>
                  </requiredResourceGroup>
            </serviceFeature>
            ...
```

Die Datei kann über den *AssetManager* mit *getAssets().open("ais.xml")* angesprochen werden. Da die Existenz einer solchen Datei alleine noch nicht garantiert, dass es sich um eine PMP-App handelt, muss mit einem XML-Parser nach einem passenden Knoten gesucht werden, z.B. <appInformation> oder <serviceFeature>.

Ein potentieller Nachteil dieser Methode ist, dass PMP-Apps, die keinerlei besondere Berechtigungen erfordern, nicht zwingend eine solche Datei benötigen. Gleichzeitig ist dies aber im vorliegenden Fall nicht bedenklich, da der Gatekeeper eine solche App nicht unbedingt berücksichtigen muss. Viel problematischer ist, dass der Gatekeeper auch hier bewusst getäuscht werden könnte, indem den Assets einfach eine formal passende Datei hinzugefügt wird.

### 5.1.4 Existenz klassischer Permissions

Alle bisher vorgestellten Ansätze sind in der Lage, tatsächliche PMP-Apps positiv zu identifizieren. Würde die PMP aber als Alternative zum bestehenden Android-Berechtiungssystem implementiert, müsste unter allen Umständen verhindert werden, dass Apps ohne Zustimmung des Anwenders uneingeschränkt daran vorbei auf das Application-Framework zugreifen dürfen. Möglich wäre dies, indem dem Gatekeeper wie zuvor beschrieben vorgetäuscht wird, dass es sich um eine PMP-App handelt, während gleichzeitig weiterhin klassische Permissions verwendet werden.

Daher muss bei der Identifizierung von Legacy-Apps ebenfalls überprüft werden, ob entsprechende <uses-permission>-Einträge im Manifest vorhanden sind. Das PackageInfo-Objekt einer App enthält unter *requestedPermissions* eine Liste dieser Einträge. Nur wenn diese Liste tatsächlich leer ist, darf eine App vom Gatekeeper ignoriert werden. Ansonsten muss der Anwender die Möglichkeit haben, diese Permissions zu steuern, auch dann, wenn es sich per Definition um eine PMP-App handelt.

### 5.1.5 System-Apps

Mit den zuvor beschriebenen Methoden ist es möglich, PMP-Apps zu identifizieren und dadurch im Umkehrschluss Legacy-Apps zu erkennen. Um allerdings eine fehlerfreie Ausführung des Android-Betriebssystems zu gewährleisten, dürfen Berechtigungen von gewissen System-Apps nicht einfach blockiert werden.

Um System-Apps zu identifizieren, kann deren *ApplicationInfo* auf die Konstanten FLAG_SYSTEM und FLAG_UPDATED_SYSTEM_APP hin untersucht werden. FLAG_SYSTEM prüft hierbei, ob eine App in der Systempartition installiert ist, FLAG_UPDATED_SYSTEM_APP dagegen ist für Apps gesetzt, die als Update einer System-App installiert wurden.

In den meisten Fällen funktioniert diese Methode. Da auf einem normalen Gerät keine Apps durch den Anwender in der Systempartition installiert werden können, gibt es keine falsch positive Identifizierung. Auf gerooteten Geräten ist es dagegen unmöglich festzustellen, ob eine App tatsächlich eine echte System-App ist, da ein Verschieben von Apps aus der und in die Systempartition mit Apps wie */system/app mover*[1] möglich ist.

Generell ist es jedoch nicht unbedingt sinnvoll, System-Apps automatisch alle Berechtigungen zu gewähren. In der Praxis werden auf Geräten mit von Herstellern angepassten Betriebssystemen eine große Anzahl Apps vorinstalliert, die vom Anwender nicht entfernt werden können (z.B. die Twitter-App auf dem Samsung Galaxy Note 2014). Diese Apps liegen ebenfalls in der Systempartition, haben aber mit der korrekten Funktionalität des Betriebssystems nichts zu tun. Daher wäre es sinnvoller, wenn zumindest bei manchen ”System-Apps” die Berechtigungen doch einschränkbar wären.

Hierbei könnte einerseits die Entscheidung dem Anwender als ”Experten” selbst überlassen werden (wodurch bei falschem Vorgehen das System beeinträchtigt werden kann); andererseits wäre es denkbar, eine Liste unproblematischer vorinstallierter Apps zu definieren, deren Einschränkung für das System keine negativen Auswirkungen hat.

## 5.2 Widerruf von Berechtigungen

Nachdem der Gatekeeper in der Lage ist, vom Anwender installierte Legacy-Apps zu identifizieren, müssen Zugriffe auf private Daten bei widerrufenen Berechtigungen abgefangen werden können. Je nachdem, auf welchem Level des Betriebssystems die PMP integriert ist, gibt es verschiedene Ansätze, dies zu bewerkstelligen.

[1] App zum Verschieben installierter Apps zwischen System- und Anwenderverzeichnis, http://goo.gl/eFoiHh

### 5.2.1 Veränderung des App-Manifests

Für die Version der PMP, die nur auf dem Application-Level arbeitet, gibt es keine Möglichkeit, direkt in die Kommunikation zwischen anderen Apps und dem Application-Framework einzugreifen. Wie in Kapitel 3.1 beschrieben wäre dies nur möglich, indem der Code der jeweiligen App durch eine Monitor-Komponente ergänzt wird.

Systeme wie Aurasium und AppGuard basieren darauf, über IRM in die Berechtigungsanfragen der Apps einzugreifen und nach benutzerdefinierten Richtlinien zu entscheiden, ob diese erlaubt werden oder nicht. Die PMP dagegen zielt darauf ab, eine eigene Art von Rechtemanagement an Stelle der Android-Permissions zu verwenden.

Um zu verhindern, dass Legacy-Apps freien Zugriff auf das System erhalten, können die entsprechenden Berechtigungen aus dem Manifest des APK entfernt und die App danach neu installiert werden. Dies würde allerdings ein Neupacken und dementsprechend ein Neusignieren der APK mit einer Software wie z.B. *apktool* erfordern.

Wie bei allen IRM-Ansätzen besteht hierbei aber das Problem der veränderten App-Signatur, daher können Updates nicht mehr direkt aus dem AppStore installiert werden. Außerdem kann es passieren (je nachdem, wie gut Fehler vom Entwickler abgefangen wurden), dass die App bei Ausführung einer Methode, die bestimmte Rechte erfordert, einfach abstürzt, wenn die entsprechende Berechtigung nicht gewährt wurde.

Insgesamt ist dieser Ansatz für den Anwender eher umständlich, da für jede veränderte App-Berechtigung das entsprechende APK neu angepasst und installiert werden muss. Wünschenswert wäre daher eine Methode, die eine Anpassung der Berechtigungen im laufenden Betrieb ermöglicht.

### 5.2.2 Manipulation der Berechtigungen im Hauptspeicher

Die Überprüfung, welche Rechte einer App durch den Anwender gewährt wurden, geschieht auf Basis der APKs der installierten Apps. Hierbei geht das System davon aus, dass bei Installation alle in deren Manifest geforderten Berechtigungen vom Anwender bestätigt wurden, da sonst die App nicht hätte installiert werden können. Die jeweiligen Berechtigungen werden in den Hauptspeicher geschrieben und überprüft, wann immer eine App eine entsprechende Methode ausführen will.

Da dieser Speicher beim Ausschalten eines Geräts geleert wird, wird der aktuelle Systemzustand laufend in verschiedenen Dateien gesichert. Beim Start werden diese Daten wiederum in den Hauptspeicher eingelesen. Die Informationen über die Berechtigungen installierte Apps liegen unter *data/system/packages.xml* (siehe Lst. 5.4), für den Zugriff darauf sind allerdings Root-Rechte erforderlich.

Eine Idee, Berechtigungen auf einem gerooteten System zu widerrufen, besteht also darin, die entsprechenden Zeilen aus dieser Datei zu entfernen und anschließend das Gerät neu zu starten [Bir09]. Allerdings ist dies keine permanente Lösung, da hierbei lediglich der aktuelle Systemzustand manipuliert und dem Gerät so vorübergehend vorgetäuscht wird, dass bestimmte Berechtigungen vom Anwender nicht gewährt wurden.

**Listing 5.4** Berechtigungen einer App in data/system/packages.xml (Auszug)

```
<package name="com.google.android.apps.translate"
    codePath="/mnt/asec/com.google.android.apps.translate-1/pkg.apk"
    nativeLibraryPath="/mnt/asec/com.google.android.apps.translate-1/lib" flags="262144"
    ft="146c41f6ba0" it="13973e3a2f1" ut="146c41f893e" version="30000060" userId="10065"
    installer="com.android.vending">
  <sigs count="1">
    <cert index="9" />
  </sigs>
  <perms>
    <item name="android.permission.READ_SMS" />
    <item name="android.permission.CAMERA" />
    <item name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <item name="android.permission.INTERNET" />
    ...
  </perms>
</package>
```

Eine weitere Einschränkung dieses Ansatzes ist, dass nicht alle Berechtigungen von Apps in derselben Form aufgeführt sind. Vorinstallierte Apps, die im Systemordner liegen, sind dort zwar mit aufgelistet, haben aber keine Berechtigungs-Parameter. Dies liegt einerseits daran, dass viele Berechtigungen in Android über den <permission>-Parameter *protectionLevel="system"* für System-Apps automatisch gewährt werden (vgl. [Theb]), andererseits werden die Berechtigungen von System-Apps teilweise auch über *<shared-user>*-Einträge kollektiv am Ende der Datei aufgeführt (siehe Lst. 5.5), weitere wiederum liegen in komplett anderen Dateien.

**Listing 5.5** <shared-user>-Berechtigungen in data/system/packages.xml (Auszug)

```
<shared-user name="android.uid.calendar" userId="10052">
  <sigs count="1">
    <cert index="1" />
  </sigs>
  <perms>
    <item name="android.permission.USE_CREDENTIALS" />
    <item name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <item name="android.permission.GET_ACCOUNTS" />
    <item name="android.permission.READ_SYNC_STATS" />
    ...
  </perms>
</shared-user>
```

Insgesamt ist dieser Ansatz nicht sinnvoll, da es für die Permissions verschiedener Arten von Apps keine einheitliche Speicherstruktur gibt. Die Sicherungsdatei wird außerdem bei jeder Änderung der Daten im Hauptspeicher (beispielsweise nach Installation oder Update einer App) neu erzeugt, was dazu führt, dass der Anwender nach einer erneuten Anpassung der Datei durch den Gatekeeper sein Gerät neu starten müsste, damit die Einschränkung der Berechtigungen erhalten bleibt.

### 5.2.3 Blockade durch das Application-Framework

Falls die PMP als Teil des Android-Betriebssystems implementiert wurde, können Zugriffe von Legacy-Apps, wie in Kapitel 3.2 beschrieben, auf verschiedenen Systemebenen abgefangen werden. Die naheliegendste Methode hierfür ist eine Erweiterung der Methode *checkPermission()* in der Klasse *ActivityManagerService*. Da alle Permission-Checks des Systems bei dieser Methode landen, muss nur an einer einzigen Stelle eine Anpassung gemacht werden.

Hierbei kann vor oder nach dem tatsächlichen Berechtigungscheck durch Android beliebiger Code ergänzt werden. Abhängig von dem Ergebnis des eigenen Berechtigungschecks muss nur eine der Konstanten PackageManager.PERMISSION_GRANTED bzw. PackageManager.PERMISSION_DENIED zurückgegeben werden.

Der Nachteil dieser Vorgehensweise ist, dass diese Methode nur Informationen über die geforderte Berechtigung und die aufrufende App hat. Während diese Informationen ausreichen, um bei einem einfachen Widerrufsmodell Entscheidungen zu treffen, können keine inhaltsbezogenen Informationen berücksichtigt werden, da nicht bekannt ist, wofür die App die Berechtigung im jeweiligen Fall tatsächlich benötigt.

Ebenfalls wird bei diesem Ansatz nicht verhindert, dass die App bei Verweigerung einer Berechtigung abstürzt. Dies würde passieren, da Entwickler in der Regel nicht davon ausgehen, dass eine bestimmte Berechtigung zur Laufzeit nicht gewährt wird.

### 5.2.4 Widerrufs-Richtlinien

Die einfachste Methode, um Sicherheit vor Zugriffen von Legacy-Apps zu garantieren, ist, einfach alle Berechtigungsanfragen durch diese Apps zu verweigern. Dies hat jedoch in den meisten Fällen zur Folge, dass die entsprechende App überhaupt nicht mehr funktioniert.

Weitaus sinnvoller ist der Ansatz, vereinzelte Berechtigungen zu gewähren, der auch von den meisten alternativen Berechtigungssystemen für Android verfolgt wird. Hierbei kann der Anwender selbst entscheiden, welche Zugriffe der App erlaubt werden sollen.

Eine Liste der vorhandenen Berechtigungen kann aus der Menge der <permission>-Elemente im Manifest der jeweiligen App erzeugt werden. Eine einfache Berechtigungssteuerung durch den Anwender würde so aussehen, dass einzelne Elemente dieser Liste gewährt werden können, wodurch die entsprechende App eingeschränkt lauffähig wäre.

Diese Steuerung könnte nach Vorlage bestehender Systeme beliebig verfeinert werden, allerdings ist dies nicht Sinn des Gatekeepers, da die PMP über ihre *serviceFeatures* selbst bereits eine genaue Einteilung der benötigten Berechtigungen abhängig von den jeweils gewünschten Funktionen der App enthält und darauf abzielt, dass Apps diese Form des Rechtemanagements verwenden. Bei Legacy-Apps ist es daher lediglich notwendig, dem Anwender die Möglichkeit zu bieten, vertrauenswürdigen Apps einzelne Berechtigungen gewähren zu können.

## 5.3 Implementierung des Prototyps

Im folgenden wird der Prototyp des Gatekeepers vorgestellt, der als *Proof-of-Concept* auf Basis des bestehenden PMP-Prototyps implementiert wurde. Abbildung 5.1 zeigt eine Übersicht der im Rahmen dieser Arbeit neu hinzugefügten und angepassten Komponenten der PMP.

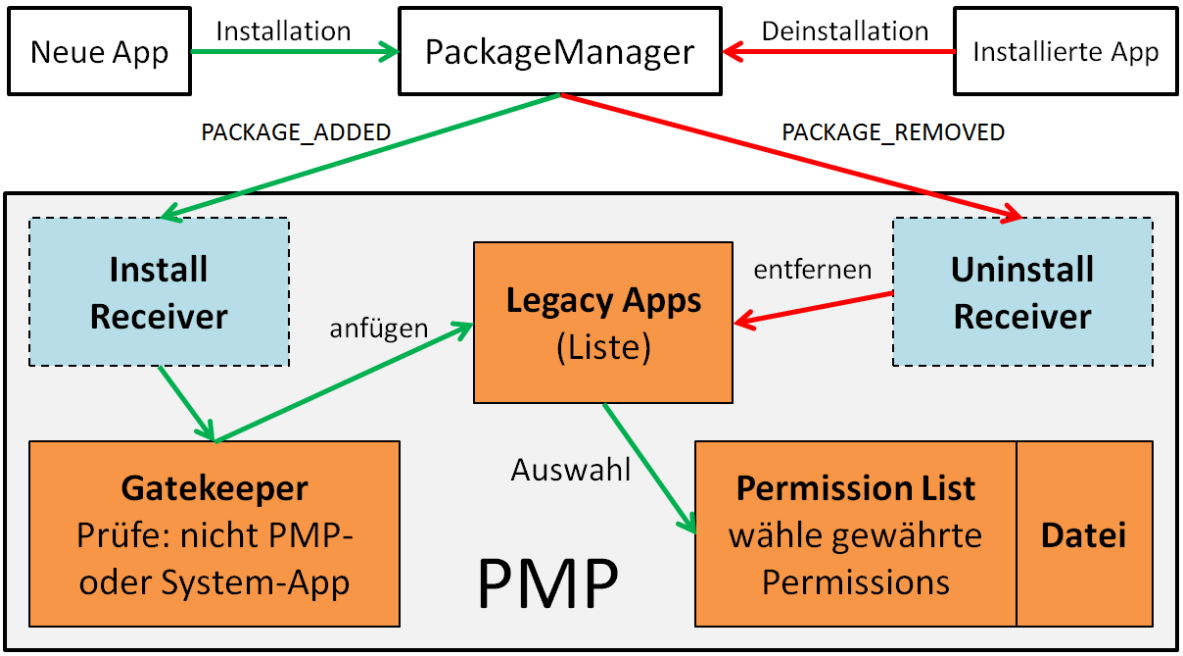



**Abbildung 5.1:** Gatekeeper: neue (orange) und angepasste (blau) Komponenten der PMP

### 5.3.1 Schnittstelle zur PMP

Das Android Application Framework reagiert auf verschiedene Systemereignisse (z.B. Empfang einer SMS, Änderung des Netzwerk- oder Bluetooth-Zustands, Aktionen des PackageManagers) mit dem Senden so genannter *Broadcasts*. Diese können von jeder App über entsprechende *BroadcastReceiver* aufgefangen und weiterverarbeitet werden.

Der bestehende PMP-Prototyp implementiert unter anderem einen *InstallReceiver* und einen *UninstallReceiver*, die auf das Hinzufügen und Entfernen von Apps über den PackageManager reagieren und den Namen des entsprechenden APKs weitergeben.

Die Schnittstelle zur neuen Gatekeeper-Komponente wurde am Ende des InstallReceivers angefügt. Hierbei wird zunächst überprüft, ob es sich um eine System-App handelt und wenn nicht, ob klassische Permission-Elemente vorhanden sind. Die entsprechenden Apps werden der Liste ”Legacy-Apps” des Gatekeepers hinzugefügt. Auch Legacy-Apps, die keine Berechtigungen anfordern, werden der Vollständigkeit halber in diese Liste mit aufgenommen.

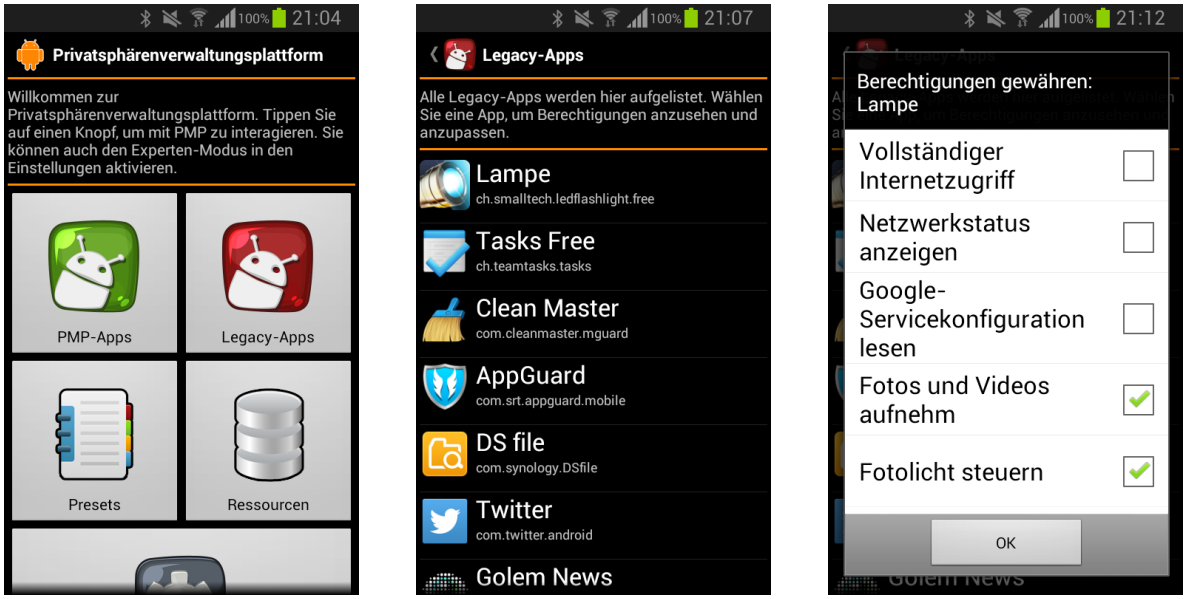
### 5.3.2 Berechtigungsmanagement

Eine auf Basis der PMP gebaute Version des Android-Betriebssystems verweigert standardmäßig alle Berechtigungen, die auf Basis von klassischen Android-Permissions angefragt werden. Da Legacy-Apps damit zum Großteil unbenutzbar werden, ermöglicht der neue Gatekeeper dem Anwender, vereinzelte Berechtigungen doch zu gewähren.

Analog zur bestehenden Liste der PMP-Apps wurde ein neuer Menüpunkt ”Legacy-Apps” hinzugefügt (siehe Abb. 5.2a), unter dem alle vom Gatekeeper entsprechend identifizierten Apps aufgelistet werden (siehe Abb. 5.2b). Mit einem Klick auf die entsprechende App wird die Liste der geforderten Berechtigungen aufgerufen, die der Anwender einzeln gewähren kann (siehe Abb. 5.2c).

Derselbe Dialog wird ebenfalls angezeigt, wenn eine neue App installiert wurde, die klassische Berechtigungen anfordert. Dadurch wird der Anwender unmittelbar darüber informiert, dass Teile der soeben installierten App möglicherweise durch den PMP-Gatekeeper blockiert werden. Hier kann der Anwender nun entscheiden, ob ausgewählte Berechtigungen gewährt werden sollen. Danach kann er testen, ob die App mit der aktuellen Konfiguration wie gewünscht funktioniert und wenn nicht, die Berechtigungen über die PMP wiederum anpassen.

Alle vom Anwender festgelegten Richtlinien werden im internen Dateisystem der PMP abgelegt. Hierbei wird neben dem *PackageName* der jeweiligen App nur eine Liste der tatsächlich gewährten Permissions gespeichert. Dies hat den Vorteil, dass auch bei einem Update der entsprechenden App bereits gewährte Permissions erhalten bleiben, gleichzeitig muss die Datei zunächst nicht angepasst werden, um etwaige durch das Update neu hinzugekommene Berechtigungen zu ergänzen.




**(a)** Hauptfenster der PMP **(b)** Liste aller Legacy-Apps **(c)** Berechtigungen gewähren

**Abbildung 5.2:** Implementierung des Gatekeepers als Teil des PMP-Prototyps

### 5.3.3 Erweiterung des Frameworks

Wie zuvor beschrieben, ist der einzige in der Praxis sinnvolle Ansatz, Berechtigungen zu widerrufen, über eine Erweiterung des Application-Frameworks. Die im Rahmen dieser Arbeit entwickelte Gatekeeper-Komponente wurde auf Basis des Application-Level PMP-Prototyps gebaut, daher war es nicht möglich, diesen Teil direkt zu implementieren.

Wie in Kapitel 5.2.3 beschrieben, kann der entsprechende Code innerhalb der Methode *checkPermission()* des ActivityManagerService ergänzt werden. Der Berechtigungscheck der PMP kann hierbei entweder vor oder nach der bestehenden Prüfung durch das Android-System stattfinden. Ein Beispiel hierfür wird in Listing 5.6 gezeigt.

Hierbei ist außerdem notwendig, aus der übergebenen UID die zugehörigen Paketnamen zu ermitteln, dies geht mit der Methode *PackageManager.getPackagesForUid()*. Die neu hinzugefügte Methode *checkGatekeeper()* muss dann lediglich prüfen, ob die angegebenen Paketnamen in der innerhalb der PMP gespeicherten Datei vorhanden und die entsprechenden Berechtigungen gewährt sind.

**Listing 5.6** Erweiterung von ActivityManagerService.checkPermission()

```
checkPermission(String permission, int pid, int uid) {
      if (permission == null)
            return PackageManager.PERMISSION_DENIED;
      }

      if (checkComponentPermission(permission, pid, uid, -1) ==
           PackageManager.PERMISSION_GRANTED) {
            String[] packageNames = PackageManager.getPackagesForUid(uid);
            return checkGatekeeper(packageNames, permission);
      } else
            return PackageManager.PERMISSION_DENIED;
}
```

### 5.3.4 Ergebnis

Der im Rahmen dieser Arbeit entwickelte Prototyp der Gatekeepers ist im Kombination mit einer geringfügigen Anpassung des Android-Frameworks in der Lage, alle durch klassische Permissions gewährten Berechtigungen von Apps zu blockieren und dadurch unerwünschte Zugriffe auf private Daten zu unterbinden. Durch eine Erweiterung der PMP können vertrauenswürdigen Apps einzelne Berechtigungen wieder gewährt werden.

Auf einem Gerät, das die PMP mitsamt der neuen Gatekeeper-Komponente als alternatives Berechtigungssystem implementiert, hat der Anwender somit volle Kontrolle über alle Zugriffe auf geschützte Daten und kann frei entscheiden, welche Informationen er den jeweiligen Apps zur Verfügung stellen möchte. Da die gewährten Berechtigungen zur Laufzeit beliebig angepasst werden können, kann der Anwender unmittelbar nachvollziehen, welche Auswirkungen einzelne Änderungen auf die Funktionalität der jeweiligen App haben. Somit kann im Laufe der Zeit die bestmögliche Balance zwischen Sicherheit und Funktionalität ermittelt werden.

# 6 Zusammenfassung und Ausblick

In der heutigen Zeit werden mobile Geräte in fast allen Bereichen des täglichen Lebens verwendet und speichern eine große Menge persönlicher Informationen. Android, das meistbenutzte mobile Betriebssystem, lässt Anwendern relativ viel Freiheit, gleichzeitig bedeutet dies aber, dass sie mehr Verantwortung bezüglich Sicherheit und Datenschutz übernehmen müssen. Allerdings sind insbesondere unerfahrene Anwender meist nicht dazu in der Lage, die tatsächlichen Risiken realistisch zu bewerten.

Um private Daten vor möglichem Missbrauch zu schützen, benötigen Apps für den Zugriff darauf besondere Rechte. Bei der Installation werden diese dem Anwender präsentiert, dabei kann aber lediglich entschieden werden, alle geforderten Rechte auf Dauer zu gewähren oder die App überhaupt nicht zu installieren. Da Apps häufig sehr viele Rechte anfordern und meistens auch Internet benötigen, geht der Anwender durch jede Installation ein Risiko ein.

Android bietet auch erfahrenen Anwendern von Haus aus keine Möglichkeit, weiteren Einfluss auf das Rechtemanagement ihrer Apps zu nehmen. Aus diesem Grund sind in den letzten Jahren verschiedene Konzepte für alternative Berechtigungssysteme entstanden.

Auf herkömmlichen Geräten können installierte Apps nur durch Manipulation ihres Quellcodes selbst eingeschränkt werden, allerdings besteht dabei keine Erfolgsgarantie, auch ist dieses Vorgehen aus rechtlicher Sicht nicht zulässig. Über eine Anpassung des Betriebssystems auf einem gerooteten Gerät ist es dagegen möglich, einzelne Berechtigungen von Apps zu widerrufen.

Bei der Privacy Management Platform sind Berechtigungen nicht direkt Apps, sondern einzelnen Features zugewiesen, die jederzeit aktiviert bzw. deaktiviert werden können. Hierbei werden alle benötigten Ressourcen vom Gerät über die PMP bezogen. PMP-kompatible Apps benötigen daher keinen direkten Zugriff auf das Application-Framework.

Je nach Implementierungsstrategie der PMP werden die Berechtigungen von Legacy-Apps entweder überhaupt nicht beeinflusst oder vollständig blockiert. Beides ist in der Praxis nicht akzeptabel, da sowohl der Schutz privater Daten als auch die Verwendung klassischer Apps möglich sein müssen. Zu diesem Zweck wurde im Rahmen dieser Arbeit der ”Gatekeeper” als neue PMP-Komponente konzipiert, der Anwendern erlaubt, vertrauenswürdigen Legacy-Apps ausgewählte Berechtigungen zu gewähren.

Der Gatekeeper muss hierbei in der Lage sein, PMP- und Legacy-Apps zuverlässig zu identifizieren, dies kann über das Vorhandensein von PMP-Schnittstellen bzw. klassischer Permission-Elemente geschehen. Ebenfalls müssen Zugriffe, deren Berechtigungen vom Anwender widerrufen wurden, verhindert werden, wobei System-Apps aus Stabilitätsgründen nicht eingeschränkt werden dürfen. Hierbei kann entweder die jeweilige App selbst manipuliert werden oder eine Blockade durch das System stattfinden.

Schließlich wurde der Gatekeeper auf Basis des bestehenden PMP-Prototyps implementiert. Nach Installation einer Legacy-Apps erhält der Anwender eine Übersicht aller geforderten Berechtigungen und kann diese individuell gewähren. Diese Einstellungen können über die PMP jederzeit angepasst werden. Damit der Widerruf der hierbei nicht gewährten Berechtigungen tatsächlich funktioniert, muss eine geringfügige Anpassung am Application-Framework vorgenommen werden.

Der im Rahmen dieser Arbeit konzipierte Gatekeeper ist also in der Lage, von Anwendern installierte Legacy-Apps sicher zu identifizieren und deren Zugriffe auf das Application-Framework zu unterbinden. Auch scheinbare PMP-Apps, die sich über klassische Permissions unberechtigten Zugriff verschaffen könnten, werden erkannt und abgefangen. Über die neue Erweiterung der PMP können Apps einzelne Zugriffsrechte wieder gewährt werden. Somit hat der Anwender volle Kontrolle darüber, auf welche Daten installierte Apps jeweils zugreifen dürfen.

## Ausblick

Mit der im Rahmen dieser Arbeit erweiterten Version der PMP kann ein Anwender frei festlegen, welche Informationen jeder App zur Verfügung stehen. Dadurch können vertrauenswürdige Legacy-Apps auch innerhalb eines auf der PMP basierenden Android-Betriebssystems weiterhin verwendet werden.

Die hierbei vorgestellte Anpassung des Application-Frameworks ist nur ein Beispiel dafür, wie Zugriffe auf geschützte Daten auf Systemebene effektiv blockiert werden können. Ein Nachteil dieses Ansatzes ist (wie in Kapitel 5.2.3 beschrieben), dass die meisten Apps einfach abstürzen, wenn eine zur Laufzeit geforderte Berechtigung nicht gewährt wurde.

Nach Vorlage der vorgestellten alternativen Berechtigungssysteme könnte man dieses Problem lösen, indem nicht der Berechtigungscheck selbst angepasst wird, sondern stattdessen die Methoden, die die eigentlichen Daten zurückgeben. Wie in Kapitel 2.3 beschrieben, rufen diese Methoden selbst den Berechtigungscheck des Systems auf, direkt danach könnte die Berechtigungsprüfung auf Basis des Gatekeepers stattfinden.

Der Vorteil hierbei wäre, dass diese Methoden nicht über SecurityExceptions das System unterbrechen, sondern tatsächliche Rückgabeparameter haben und somit bei einer durch die PMP widerrufenen Berechtigungen einen leeren (aber passenden) Datensatz an die App zurückgeben könnten, wodurch das System normal weiterarbeiten würde. Unberechtigte Internet-Anfragen könnten wiederum durch eine Offline-Meldung abgefangen werden. Allerdings müsste bei diesem Ansatz jede einzelne Methode im Application-Framework, die Zugriff auf durch Permissions geschützte Quellen hat, angepasst werden, was vergleichsweise aufwändig ist.

Durch eine derartige Erweiterung des Gatekeepers würde die Zuverlässigkeit von Legacy-Apps in der Praxis erhöht werden, da zur Laufzeit auftretende Zugriffe auf geschützten Daten blockiert werden könnten, ohne dass der Programmfluss unterbrochen wird. Dadurch wären auch stark eingeschränkte Apps weiterhin lauffähig und Anwender könnten frei entscheiden, welche Informationen der App tatsächlich zur Verfügung gestellt werden sollen.

# Literaturverzeichnis


[AZHL12] K. W. Y. Au, Y. F. Zhou, Z. Huang, D. Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12. 2012. (Zitiert auf Seite 15)

[BCG13] K. Benton, L. Camp, V. Garg. Studying the effectiveness of android application permissions requests. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, PerCom '13. 2013. (Zitiert auf Seite 13)

[BGH+13a] M. Backes, S. Gerling, C. Hammer, M. Maffei, P. von Styp-Rekowsky. AppGuard - Fine-grained Policy Enforcement for Untrusted Android Applications. Technischer Bericht A/02/2013, Saarland University, 2013. (Zitiert auf Seite 25)

[BGH+13b] M. Backes, S. Gerling, C. Hammer, M. Maffei, P. von Styp-Rekowsky. AppGuard: Enforcing User Requirements on Android Apps. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13. 2013. (Zitiert auf Seite 25)

[Bir09] T. Bird. Changing application security permissions after installation. Wiki-Eintrag auf elinux.org, 2009. URL `http://elinux.org/Android_Security`. (Zitiert auf Seite 45)

[BRSS11] A. R. Beresford, A. Rice, N. Skehin, R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11. 2011. (Zitiert auf den Seiten 30 und 31)

[CCFZ12] M. Conti, B. Crispo, E. Fernandes, Y. Zhauniarovich. CRePE: A System for Enforcing Fine-Grained Context-Related Policies on Android. *Information Forensics and Security, IEEE Transactions on*, 7(5):1426–1438, 2012. (Zitiert auf Seite 29)

[DC13] B. Davis, H. Chen. RetroSkeleton: Retrofitting Android Apps. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13. 2013. (Zitiert auf Seite 23)

[dev14] developer.android.com. Application Fundamentals, 2014. URL `http://developer.android.com/guide/components/fundamentals.html`. (Zitiert auf Seite 17)

[DSKC12] B. Davis, B. Sanders, A. Khodaverdian, H. Chen. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *IEEE Mobile Security Technologies*, MoST'12. 2012. (Zitiert auf Seite 23)

[EGC+10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10. 2010. (Zitiert auf den Seiten 14 und 32)

[Erd13] F. Erdle. Android: Die Geschichte des Erfolgs. connect.de, 2013. URL http://www.connect.de/ratgeber/android-geschichte-des-erfolgs-1491130.html. (Zitiert auf Seite 9)

[FCH+11] A. P. Felt, E. Chin, S. Hanna, D. Song, D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11. 2011. (Zitiert auf den Seiten 13 und 15)

[FEW12] A. P. Felt, S. Egelman, D. Wagner. I've Got 99 Problems, but Vibration Ain't One: A Survey of Smartphone Users' Concerns. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12. 2012. (Zitiert auf Seite 13)

[FHE+12] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, D. Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12. 2012. (Zitiert auf Seite 10)

[Fri12] C. Frickel. Google löscht Anti-Schnüffel-App für Android, 2012. URL http://goo.gl/DkUsSR. (Zitiert auf Seite 26)

[goo14] google.com. App-Berechtigungen prüfen, 2014. URL https://support.google.com/googleplay/answer/6014972?hl=de. (Zitiert auf Seite 13)

[HHJ+11] P. Hornyack, S. Han, J. Jung, S. Schechter, D. Wetherall. These Aren'T the Droids You'Re Looking for: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11. 2011. (Zitiert auf Seite 32)

[HSD13] H. Hao, V. Singh, W. Du. On the Effectiveness of API-level Access Control Using Bytecode Rewriting in Android. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13. 2013. (Zitiert auf Seite 21)

[Izz14] Izzy. Why do so many applications require permission to read the phone state and identity?, 2014. URL http://goo.gl/jtciYF. (Zitiert auf Seite 16)

[JMV+12] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12. 2012. (Zitiert auf Seite 22)

[Kö14] T. Költzsch. Android läuft auf fast 85 Prozent aller Smartphones. golem.de, 2014. URL http://goo.gl/usW4sP. (Zitiert auf Seite 9)

[Kum12] S. Kumar. Architecture of Android, 2012. URL http://androidprogramz.blogspot.de/2012/06/architecture-of-android-in-order-to.html. (Zitiert auf Seite 17)

[NKZ10] M. Nauman, S. Khan, X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10. 2010. (Zitiert auf den Seiten 27 und 28)

[Pan13] A. Panos. *BYOD - Private Hardware in der Firma nutzen.* Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Deutschland, 2013. (Zitiert auf Seite 37)

[Ros13] S. Rosenblatt. Why Android won't be getting App Ops anytime soon. Auf den Seiten von cnet, 2013. URL http://www.cnet.com/news/why-android-wont-be-getting-app-ops-anytime-soon/. (Zitiert auf Seite 19)

[Rud13] D. Ruddock. Google Play Services 4.0 Requires Developers To Use The New "Advertising ID" To Identify Your Device, Enforcement Starts Aug 2014, 2013. URL http://goo.gl/pfMZD8. (Zitiert auf Seite 16)

[SB11] C. Stach, A. Brodt. — vHike — A Dynamic Ride-sharing Service for Smartphones. In *Proceedings of the 12th International Conference on Mobile Data Management*, MDM'11. 2011. (Zitiert auf Seite 10)

[SC13] J. Sellwood, J. Crampton. Sleeping Android: The Danger of Dormant Permissions. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '13. 2013. (Zitiert auf Seite 16)

[Sch13] P. Scholz. *Integration der PMP in das Android OS.* Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Deutschland, 2013. (Zitiert auf Seite 40)

[SLG+12] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, I. Molloy. Android Permissions: A Perspective Combining Risks and Benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12. 2012. (Zitiert auf Seite 21)

[SM13] C. Stach, B. Mitschang. Privacy Management for Mobile Platforms - A Review of Concepts and Approaches. In *Proceedings of the 14th International Conference on Mobile Data Management*, S. 1–9. IEEE Computer Society Conference Publishing Services, 2013. (Zitiert auf den Seiten 10 und 26)

[SM14] C. Stach, B. Mitschang. Design and Implementation of the Privacy Management Platform. In *Proceedings of the 15th International Conference on Mobile Data Management*, MDM'14. 2014. (Zitiert auf Seite 39)

[sou14] source.android.com. Introducing ART, 2014. URL https://source.android.com/devices/tech/dalvik/art.html. (Zitiert auf Seite 17)

[SS12] C. Stach, L. F. Schlindwein. Candy Castle - A Prototype for Pervasive Health Games. In *Proceedings of the 2012 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, PerCom'12. 2012. (Zitiert auf Seite 10)

[Sta11] C. Stach. Saving time, money and the environment - vHike a dynamic ride-sharing service for mobile devices. In *Proceedings of the 2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, PerCom'11. 2011. (Zitiert auf Seite 10)

[Sta13a] C. Stach. How to Assure Privacy on Android Phones and Devices? In *Proceedings of the 14th International Conference on Mobile Data Management*, MDM'13. 2013. (Zitiert auf Seite 35)

[Sta13b] C. Stach. Wie funktioniert Datenschutz auf Mobilplattformen? In *Informatik 2013: Informatik angepasst an Mensch, Organisation und Umwelt, 43. GI Jahrestagung*, Lecture Notes in Informatics. 2013. (Zitiert auf den Seiten 36 und 40)

[Thea] The Android Open Source Project. ActivityManagerService.java (Android 2.3, Source Code). URL http://goo.gl/j9TJyK. (Zitiert auf Seite 19)

[Theb] The Android Open Source Project. AndroidManifest (Source Code). URL https://github.com/android/platform_frameworks_base/blob/master/core/res/AndroidManifest.xml. (Zitiert auf Seite 46)

[XSA12] R. Xu, H. Saïdi, R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12. 2012. (Zitiert auf den Seiten 23 und 24)

[ZXMX13] Y. Zhongyang, Z. Xin, B. Mao, L. Xie. DroidAlarm: An All-sided Static Analysis Tool for Android Privilege-escalation Malware. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13. 2013. (Zitiert auf Seite 14)

Alle URLs wurden zuletzt am 15. 08. 2014 geprüft.

**Erklärung**

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

---

Ort, Datum, Unterschrift