Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Ein sicherer Object Store für Android

Diana Salsa

Studiengang: Informatik

Prüfer/in: Prof. Bernhard Mitschang

Betreuer/in: Dipl.-Inf. Christoph Stach

Beginn am: 12. Januar 2017

Beendet am: 12. Juli 2017

CR-Nummer: K.4.1, K.6.5

Kurzfassung

Mobilgeräte speichern heutzutage eine große Menge persönlicher Daten. Während die Weitergabe solcher Daten zwischen Apps durch Berechtigungssysteme eingeschränkt wird und somit der Kontrolle des Anwenders unterliegt, sind unerwünschte Zugriffe durch Ausnutzung von Sicherheitslücken möglich.

Im Rahmen dieser Arbeit werden zunächst bestehende und alternative Lösungsansätze für die App-übergreifende Datenweitergabe analysiert und bewertet. Darauf aufbauend wird der *Secure Object Container* als Plattform für den Austausch beliebiger Datensätze zwischen Apps konzipiert, wobei die Sicherheit der abgelegten Daten dabei ebenso priorisiert wird wie die einfache Einbindung für App-Entwickler.

Für den Entwurf des Prototyps auf Basis des Android-Betriebssystems werden mögliche Implementierungsvarianten der einzelnen Kernkomponenten verglichen und bewertet. Der Secure Object Container wird schließlich als Teil der *Privacy Management Platform* implementiert und hinsichtlich Performance und Sicherheit analysiert.

Inhaltsverzeichnis

1 Einleitung **11**
- 1.1 Situation . . . 11
- 1.2 Problemstellung . . . 12
- 1.3 Anforderungen . . . 13

2 Verwandte Ansätze **15**
- 2.1 Native Systemfunktionen . . . 15
  - 2.1.1 Öffentliches Dateisystem . . . 15
  - 2.1.2 Intent . . . 16
  - 2.1.3 Content Provider . . . 18
  - 2.1.4 Clipboard . . . 20
- 2.2 Alternative Ansätze . . . 22
  - 2.2.1 MetaService . . . 22
  - 2.2.2 Porscha . . . 24
  - 2.2.3 Kynoid . . . 26
  - 2.2.4 Secure Data Container . . . 27
- 2.3 Zusammenfassung . . . 29

3 Der Secure Object Container **31**
- 3.1 Schnittstellen . . . 31
  - 3.1.1 Datenverwaltung . . . 32
  - 3.1.2 Kill Switch . . . 33
  - 3.1.3 API-Wrapper . . . 33
- 3.2 Schema . . . 34
- 3.3 Verschlüsselung . . . 35
- 3.4 Sicherheitsanalyse . . . 36
  - 3.4.1 Physikalischer Angriff . . . 36
  - 3.4.2 Softwarebasierter Angriff . . . 37

4 Implementierung **39**
- 4.1 Service-Interface . . . 39
- 4.2 Serialisierung von Objekten . . . 41
  - 4.2.1 Native Methoden . . . 41
  - 4.2.2 Implementierungsunabhängige Ansätze . . . 42
  - 4.2.3 Fazit . . . 44

4.3 Datenspeicher . . . 44
4.3.1 Shared Preferences . . . 45
4.3.2 Internes Dateisystem . . . 45
4.3.3 Datenbank . . . 45
4.4 Verschlüsselung . . . 48
4.4.1 Verfahren . . . 48
4.4.2 Wahl des Schlüssels . . . 49
4.4.3 Schlüssel sicher ablegen . . . 49
4.4.4 Sicheres Löschen . . . 51
4.5 SOC-API . . . 52
4.5.1 Basisfunktionalität . . . 52
4.5.2 User Interface . . . 53
4.5.3 Fazit . . . 54

5 Prototyp **55**
5.1 Die Privacy Management Platform . . . 55
5.1.1 Komponenten . . . 55
5.1.2 Technischer Aufbau . . . 57
5.2 Implementierung des Protoyps . . . 60
5.2.1 Ressource . . . 60
5.2.2 PMP-API . . . 62
5.2.3 Gesamtübersicht . . . 64

6 Evaluation **65**
6.1 Serialisierung . . . 65
6.2 SOC-Testapp . . . 68
6.3 Geschwindigkeit . . . 69
6.4 Speicherverbrauch . . . 71
6.5 Erfüllung der Anforderungen . . . 72
6.6 SOC vs. SDC . . . 73
6.6.1 Vorteile des SOC . . . 73
6.6.2 Vorteile des SDC . . . 74

7 Zusammenfassung und Ausblick **75**

Literaturverzeichnis **79**

Abbildungsverzeichnis

2.1 Intent Filter . . . 17
2.2 Content Provider . . . 19
2.3 Clipboard mit mehreren Objekten . . . 21
2.4 Architektur des MetaService . . . 22
2.5 Integration des Porscha-Mediators beim SMS-Empfang . . . 24
2.6 Architektur des Kynoid-Frameworks . . . 26
2.7 Datenbankschema des Secure Data Container . . . 28

3.1 SOC-Datenoperationen aus Sicht von Besitzer und Empfänger . . . 31
3.2 Kommunikation von Apps mit dem SOC über einen API-Wrapper . . . 33
3.3 Relationales Datenbankschema des SOC . . . 34
3.4 Zugriff auf den SOC am Beispiel einer Leseoperation . . . 35

4.1 Übersicht der Hauptkomponenten des SOC . . . 39
4.2 Kommunikation zwischen Client und Remote Service via AIDL-Interface . . . 40
4.3 Beispiel einer relationalen Datenbank mit Schlüsselverweisen . . . 46
4.4 Sicherheitsarchitektur mit TrustZone-Unterstützung . . . 50
4.5 Kernkomponenten und Zugriffspunkte von SOC-API und SOC-Service . . . 53

5.1 Berechtigungsmodell der Privacy Management Platform . . . 56
5.2 Technischer Aufbau der Privacy Management Platform . . . 59
5.3 Methodenaufrufe des SOC . . . 61
5.4 Dialog-Wrapper der SOCmanagerUI . . . 63
5.5 Kernkomponenten und Zugriffspunkte des SOC-Prototyps . . . 64

6.1 Aufbau der Testklassen MedicalData und Patient . . . 65
6.2 Serialisierungsgeschwindigkeit eines einfachen Objekts . . . 66
6.3 Serialisierungsgeschwindigkeit eines komplexen Objekts . . . 66
6.4 Speicherverbrauch der verschiedenen Serialisierungsmethoden . . . 67
6.5 Benutzeroberfläche der SOC-Testapp . . . 68
6.6 Vergleich: Durchschnittliche Dauer einer Leseoperation . . . 69
6.7 Vergleich: Durchschnittliche Dauer einer Schreiboperation . . . 70
6.8 Vergleich: Durchschnittlicher Speicherverbrauch pro Objekt . . . 71

Tabellenverzeichnis

2.1 Vergleich der betrachteten Ansätze hinsichtlich der Anforderungen 29

6.1 SQLite-Performance der verwendeten Testgeräte 70

Verzeichnis der Listings

4.1 Serialisierung mit Simple XML 43
4.2 Serialisierung mit GSON 44

5.1 Registrierungsaufruf einer PMP-App 57
5.2 Serviceaufruf einer PMP-App 57
5.3 Informationsset einer PMP-App 58
5.4 Informationsset einer PMP-Ressourcengruppe 58
5.5 Interface der SOC-Ressource 60
5.6 Interface zur Rückgabe der SOC-Daten 63

1 Einleitung

Mobilgeräte sind heutzutage allgegenwärtig. Statistiken zeigen, dass 2016 nahezu 50 Millionen Menschen in Deutschland ein Smartphone nutzten und diese Zahl mit jedem Jahr weiter zunimmt [Sta17b]. Dies entspricht ca. 60% der Gesamtbevölkerung, wobei der Anteil bei Menschen im Alter von 12 bis 60 Jahren noch deutlich höher ausfällt.

Im Gegensatz zu traditionellen „Handys“ handelt es sich hierbei um komplexe und leistungsstarke Computersysteme, deren Anwendungsbereich weit über das Telefonieren und Schreiben von Nachrichten hinausgeht. Unterstützt wird dies durch das nahezu unüberschaubare Angebot von Anwendungen (kurz: *Apps*), die Anwendern über zentrale *App-Stores* zur Verfügung gestellt werden. Die dort verfügbaren Apps stammen von einer breiten Masse von Drittanbietern, die Lösungen für jeden erdenklichen Bedarf anbieten. Dies ermöglicht Anwendern, durch deren Nutzung nahezu alle Bereiche des täglichen Lebens zu unterstützen.

1.1 Situation

Diese Art der Nutzung hat jedoch zur Folge, dass die Geräte eine große Menge persönlicher Informationen speichern. Angefangen bei Kontakt- und Kalenderdaten über Standortinformationen für Navigationssysteme bis hin zu detaillierten medizinischen Daten durch Nutzung von Fitness-Trackern – all diese Informationen werden aufgezeichnet bzw. gespeichert [Shi09].

Obwohl diese Daten nur für einen bestimmten Zweck erhoben werden, kann nicht ausgeschlossen werden, dass Apps diese ohne Wissen des Anwenders weiterverbreiten. Studien belegen, dass dies bei einer großen Anzahl der am häufigsten genutzten Apps aus dem Google Play Store zutrifft, etwa durch die Weitergabe von Daten an Werbeanbieter für personenbezogene Werbung [EGH+14].

Vorhandene Systeme bieten hierfür jedoch keine zufriedenstellenden Lösungsansätze: Sowohl bei Android als auch bei Apples iOS besteht lediglich die Möglichkeit, eine Zugriffsberechtigung auf geschützte Daten für die gesamte App anzufordern, wobei der Anwender danach keine Kontrolle mehr darüber hat, wofür diese tatsächlich genutzt wird [XSA12]. Während bei früheren Systemversionen von Android ein *all-or-nothing*-Ansatz implementiert wurde (entweder alle Berechtigungen zu gewähren oder die Installation abzubrechen), ist es heutzutage zumindest möglich, „gefährliche“ Berechtigungen individuell zu bestätigen.

Jedoch bietet auch der Schutz durch ein Berechtigungssystem keine tatsächlich Garantie dafür, dass private Daten nicht an unberechtigte Apps weitergegeben werden. Davi et al. beschreiben, wie Apps ohne entsprechende Berechtigung Zugriff auf geschützte Daten erlangen können, indem sie nicht ausreichend geschützte Komponenten einer anderen (berechtigten) App ausnutzen [DDSW10]. Demnach stellt jede App, der Zugriffsberechtigungen auf private Daten gewährt wurden, eine potentielle Sicherheitslücke dar.

In jedem Fall ist jedoch der Anwender für die Entscheidung verantwortlich, Berechtigungsanfragen von Apps zu bestätigen. Obwohl in den vergangenen Jahren das Datenschutzbewusstsein in der Bevölkerung insgesamt zugenommen hat [BJL+13], fühlen sich viele Anwender noch immer technisch damit überfordert, individuelle Einstellungen für Apps zu definieren; dabei passiert es häufig, dass Berechtigungsanfragen aus Bequemlichkeit pauschal gewährt werden [BCG13], was auch das Risiko unberechtigter Zugriffe deutlich erhöht.

Insgesamt bieten vorhandene Berechtigungssysteme daher keinen tatsächlichen Schutz vor dem Missbrauch persönlicher Daten. Ist eine böswillige App darüber hinaus in der Lage, Root-Rechte zu erlangen, kann sie direkt auf den internen Datenspeicher aller installierten Apps zugreifen (ohne „Umweg“ über das Berechtigungssystem) und damit beispielsweise unverschlüsselt gespeicherte Daten auslesen oder verändern.

1.2 Problemstellung

Basierend auf der zuvor beschriebenen Situation wäre es naheliegend, den Zugriff auf App-Daten durch Dritte vollständig zu unterbinden, um das Risiko unerwünschter Zugriffe möglichst gering zu halten. In der Realität sind die meisten Apps jedoch auf entsprechende Berechtigungen angewiesen, um korrekt funktionieren zu können. Darüber hinaus erlaubt das Abrufen benötigter Informationen im Hintergrund einen höheren Grad an Automatisierung und damit mehr Komfort für den Anwender.

Ein einfaches Beispiel hierfür ist die Verknüpfung zwischen Adressbuch und Messenger-Apps (z.B. WhatsApp). Hierbei fordert der Messenger Zugriff auf das persönliche Adressbuch an, um dieses laufend mit der eigenen Nutzerdatenbank abzugleichen. Im Gegenzug erhält der Anwender eine Übersicht aller seiner Kontakte, die über diesen Messenger erreichbar sind; im Adressbuch wird ein entsprechender Button zur Kontaktaufnahme ergänzt.

Dies ist einerseits datenschutzrechtlich extrem problematisch, da eine große Menge Kontaktdaten Dritter ungefiltert an den Messenger-Anbieter übermittelt wird, ohne dass deren Inhaber dagegen Einspruch erheben könnten; darunter möglicherweise auch vertrauliche Informationen von Geschäftskontakten. Ebenfalls erhöht eine solche Verbreitung das Risiko für die Datensicherheit enorm, da jeder Messenger-Dienst Sicherheitslücken enthalten und damit unberechtigte Zugriffe auf die Kontaktinformationen ermöglichen könnte. In jedem Fall wäre daher auch wünschenswert, die Menge geteilter Daten beliebig einzuschränken zu können, um potentielle Datenlecks möglichst gering zu halten.

Insgesamt ist also ein Ansatz erforderlich, der den sicheren Austausch von Daten zwischen Apps ermöglicht, wobei deren Besitzer volle Kontrolle darüber haben muss, welche Daten welchen Apps zur Verfügung stehen. Darüber hinaus muss garantiert sein, dass nur der beabsichtigte Empfänger die geteilte Daten abrufen kann, wobei es unberechtigten Apps weder über entsprechende Empfänger noch durch sonstige Sicherheitslücken im System möglich sein darf, Zugriff zu erhalten.

1.3 Anforderungen

Um zu verhindern, dass Apps unberechtigten Zugriff auf geschützte Informationen erlangen können, muss ein Datenaustausch-Verfahren entsprechende Sicherheitsanforderungen erfüllen. McCumber definiert Informationssicherheit über die Eigenschaften *Confidentiality*, *Integrity* und *Availability* [McC91]. Im Folgenden werden diese Begriffe für die vorliegende Problemstellung festgelegt:

- **Confidentiality (Vertraulichkeit)** - Nur berechtigte Instanzen dürfen Zugriff auf die abgelegten Daten erhalten. Neben dem Inhaber eines Datensatzes gilt dies nur für Dritte, denen der Inhaber explizit das entsprechende Zugriffsrecht eingeräumt hat. Ein Widerruf dieses Rechts muss jederzeit möglich sein.
- **Integrity (Unversehrtheit)** - Die abgelegten Daten dürfen nur von dazu berechtigten Instanzen verändert werden. Hat nur der Inhaber eines Datensatzes dieses Recht, muss garantiert sein, dass ein Verändern der Daten durch Dritte unmöglich ist.
- **Availability (Verfügbarkeit)** - Die Austauschplattform sowie alle darauf abgelegten Daten müssen zu jeder Zeit für alle Apps verfügbar sein, die entsprechende Zugriffsrechte besitzen. Sind die Apps dagegen selbst nicht verfügbar, ist die Verfügbarkeit von Plattform und Daten ebenfalls nicht erforderlich (z.B. bei ausgeschaltetem Gerät).

Um darüber hinaus die in Abschnitt 1.1 beschriebenen Defizite bestehender mobiler Berechtigungssysteme zu verbessern, ist folgende Eigenschaft notwendig:

- **Feingranulare Berechtigungen** - Zugriffsberechtigungen werden individuell zugewiesen, sowohl im Hinblick auf die Datensätze als auch bezüglich berechtiger Apps.

Damit sich das Verfahren als brauchbare Alternative gegenüber existierenden Ansätzen durchsetzen kann, muss es für eine möglichst breite Basis (prinzipiell alle existierenden Apps) nutzbar sein. Daraus ergeben sich die folgenden beiden Anforderungen:

- **Genericity (Allgemeine Anwendbarkeit)** - Die Plattform muss die Ablage beliebiger Datenstrukturen erlauben, wobei alle Zugriffe nach demselben Schema durchgeführt werden, unabhängig von der Art des Datensatzes.
- **Usability (Benutzerfreundlichkeit)** - Das Einbinden der Plattform sowie deren Nutzung zur Laufzeit muss für den Entwickler einer App nahtlos und ohne umfangreichen Implementierungsaufwand möglich sein.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Verwandte Ansätze beschreibt, welche Funktionen zum Datenaustausch vom Android-Betriebssystem bereitgestellt werden. Darüber hinaus werden alternative Lösungsansätze vorgestellt, die mehr Schutz gegen potentielle unberechtigte Zugriffe durch Dritte bieten sollen.

Kapitel 3 – Der Secure Object Container definiert das grundlegende Konzept für eine sichere Plattform zum Datenaustausch, bei der volle Kontrolle darüber besteht, welche Daten in welcher Form für Dritte freigegeben werden. Dabei werden potentielle Angriffsszenarien betrachtet und bewertet, inwieweit das vorgestellte Konzept diesen standhält.

Kapitel 4 – Implementierung stellt einen Entwurf für den technischen Aufbau des Objekt Containers auf Basis der Android-Plattform vor. Dabei werden mögliche Implementierungsvarianten für verschiedene Kernkomponenten vorgestellt und bewertet.

Kapitel 5 – Prototyp beschreibt die Realisierung des SOC-Prototyps auf Basis der Privacy Management Platform. Dabei werden zunächst die technischen Anforderungen an deren Apps und Ressourcen beleuchtet; darauf aufbauend wird erklärt, inwieweit der Protoyp die im vorhergehenden Kapitel vorgestellten Implementierungsaspekte umsetzt.

Kapitel 6 – Evaluation bewertet den Protoyp und vergleicht ihn mit bestehenden Ansätzen. Zur Demonstration wird eine Testapp entworfen, die alle vom Prototyp bereitgestellten Schnittstellen nutzen kann; dabei werden Performance und Funktionsweise denen alternativer Ansätze direkt gegenübergestellt und bewertet.

Kapitel 7 – Zusammenfassung und Ausblick fasst die Kernpunkte und das Ergebnis der Arbeit zusammen und stellt Ansätze vor, wie die Möglichkeiten des SOC durch zusätzliche Funktionen bzw. alternative Implementierungen erweitert werden könnten.

2 Verwandte Ansätze

Im Folgenden wird betrachtet, wie bestehende Systeme und Konzepte die zuvor definierten Anforderungen umsetzen. Der Fokus liegt dabei auf der einseitig gerichteten Übertragung von Informationen, das heißt die empfangende App soll ohne Wissen des Besitzers keine direkten Änderungen an den Originaldaten vornehmen können.

Hierbei wird die Android-Plattform als Beispiel gewählt, wobei die zugrundeliegenden Konzepte allgemein gültig sind und auch auf andere mobile Plattformen, wie z.B. iOS, angewendet werden können [BKOS10]. Alle Ansätze werden abschließend auf Basis der in Abschnitt 1.3 definierten Anforderungen bewertet.

2.1 Native Systemfunktionen

Zunächst werden die im Android-System bereits vorhandenen Funktionalitäten beschrieben, die zum Datenaustausch zwischen Apps eingesetzt werden können. Hierbei gibt es zwei grundlegende Varianten: persistente Speicherung mit Zugriffskontrolle durch das Berechtigungssystem oder einmalige Weitergabe über einen System-Service.

2.1.1 Öffentliches Dateisystem

Die wohl einfachste Art und Weise, beliebige Datenstrukturen mit Apps zu teilen, ist über die Ablage einer Kopie im öffentlichen Dateisystem [dev17n]. Eine dort gespeicherte Datei kann von jeder App gelesen werden, die die Berechtigung *READ_EXTERNAL_STORAGE* bzw. *WRITE_EXTERNAL_STORAGE* besitzt.

Die App muss dabei lediglich das von der Datei verwendete Format verstehen können. Hierbei kann entweder ein bekanntes Standardformat verwendet werden (deklariert durch die Dateiendung) oder ein durch Absprache zwischen den Apps (bzw. deren Entwicklern) definiertes Sonderformat (z.B. auch auf Textbasis).

Um komplexe Objekte direkt auf Dateien abzubilden, können sie in Text- oder Binärform umgewandelt (serialisiert) werden. Dies ist beispielsweise mit Objektstrukturen von Klassen möglich, die *java.io.Serializable* [Ora15] implementieren. Das Hin- und Rückkonvertieren zwischen Objekt und Datei ist hiermit ohne objektspezifischen Programmieraufwand möglich. Alternativ dazu kann das Serialisierungsschema einer Klasse auch manuell definiert werden.

In beiden Fällen muss der Empfänger die Klassendefinition des erhaltenen Objekts kennen, um die Deserialisierung durchführen zu können. Ein Austausch „unbekannter“ Objektstrukturen ist daher grundsätzlich nicht möglich.

Bewertung

Über das öffentliche Dateisystem können Datenstrukturen zwischen beliebigen Apps ausgetauscht werden, ohne dass die Apps direkt miteinander kommunizieren oder überhaupt voneinander wissen müssen.

Der große Nachteil dieses Ansatzes ist jedoch, dass jede App mit Zugriff auf das Dateisystem auch Zugriff auf *alle* dort abgelegten Daten hat. Selbst die serialisierte Form macht Daten nicht „sicher“ – auch ohne die konkrete Klassendefinition zur Deserialisierung können Inhalte ausgelesen werden, da z.B. das verwendete Binärformat öffentlich dokumentiert ist.

Um Daten vor unerwünschten Zugriffen zu schützen, kann eine Verschlüsselungsstrategie genutzt werden. Neward beschreibt, wie *javax.crypto.SealedObject* und *java.security.SignedObject* in Kombination mit traditioneller Java-Serialisierung genutzt werden können, um Daten vor unerwünschten Zugriffen zu schützen [New10]. Dadurch sind die Daten nur für Apps lesbar, die den entsprechenden Schlüssel kennen.

Trotzdem wären die (verschlüsselten) Dateien auch weiterhin für eine große Anzahl von Apps einsehbar und es kann daher nicht ausgeschlossen werden, dass Angreifer langfristig eine Möglichkeit finden, an die enthaltenen Informationen zu kommen. Problematisch ist hierbei, dass ein Austausch des Schlüssels zwischen Apps stattfinden muss, damit die Daten Dritten zur Verfügung gestellt werden können. Dabei besteht das Risiko, dass auch unberechtigte Instanzen Zugriff auf den Schlüssel und damit die freigegebenen Daten erlangen.

2.1.2 Intent

Komponenten von Android-Anwendungen (z.B. Activities) können über asynchrone Nachrichten Aktionen anderer Komponenten anfordern. Die Definition einer solchen Nachricht wird über ein *Intent*-Objekt [dev17h] formuliert, welches im Kern aus zwei Parametern besteht:

- **data** - URI eines Datensatzes (content:) bzw. Daten mit definiertem Protokoll (z.B. http:)
- **action** - Die Aktion, die auf Basis der gesendeten Daten ausgeführt werden soll

Ein einfaches Beispiel ist die Übergabe einer Telefonnummer als Datensatz (z.B. tel:12345) mit der Anweisung *ACTION_DIAL*. Sofern kein festes Ziel spezifiziert wird, wählt das System auf Basis von Datenformat und Aktion kompatible Ziel-Apps aus (im vorliegenden Beispiel die Telefon-App). Jede App definiert mithilfe von Intent-Filtern selbst, welche Arten von Anweisungen und Datentypen sie verarbeiten kann. Gibt es mehrere kompatible Apps, wird dem Anwender eine entsprechende Auswahlliste angezeigt (siehe Abbildung 2.1).

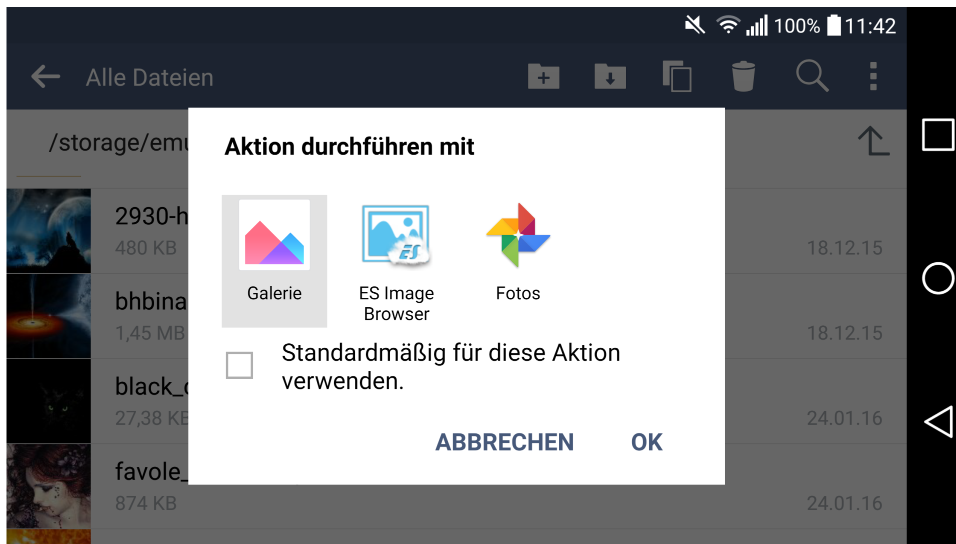


Abbildung 2.1: Auswahlliste von Apps mit kompatiblem Intent-Filter

Dem gegenüber stehen sogenannte explizite Intents, bei denen zusätzlich das Ziel der Nachricht, also die Empfänger-Komponente, spezifiziert wird. Diese können beispielsweise in Kombination mit *startActivity(Intent)* verwendet werden, um ein neues Fenster zu öffnen. Eine Activity wird hierbei direkt über ihren Klassennamen angesprochen.

Es ist dabei auch möglich, Activities externer Apps anzusprechen; hierfür muss zunächst der interne Bezeichner der Ziel-App bekannt sein (z.B. de.dianasalsa.masterarbeit). Der Aufruf ist dann entweder über den Namen einer Activity-Klasse möglich (die allerdings mit *exported = true* externe Zugriffe erlauben muss) oder über eine in der Ziel-App definierte Aktion. Beide Ansätze werden in einem Blogeintrag von Krumelur vorgestellt [Kru16].

Weitergabe von Dateiobjekten

Neben dem in *data* definierten Datensatz können zusätzliche Objekte an einen Intent-Aufruf angehängt werden. Die Methode *Intent.putExtra()* akzeptiert ein Schlüssel/Wert-Paar; die Empfänger-Activity kann den Wert entsprechend mit *Intent.getExtra()* auslesen. Für den Transfer können auch mehrere Paare innerhalb eines einzigen *Bundle*-Objekts kombiniert werden.

Auf diese Art übertragene Datensätze dürfen nur primitive Datentypen, Strings und darauf basierende Strukturen wie Listen oder Arrays enthalten [dev17g]; daher ist für den Transfer komplexer Objekte eine Serialisierung notwendig. Android bietet hierfür zwei Methoden an: *Bundle.putParcelable()* und *Bundle.putSerializable()*. Letzteres ist die automatische Standard-Serialisierung von Java, *Parcelable* erlaubt/erfordert dagegen eine explizite Definition des Marshalling-Vorgangs durch den Entwickler der betroffenen Klasse mithilfe der Methoden *writeToParcel()* und *createFromParcel()* [dev17j].

Bewertung

Durch die Verwendung von Intents ist es möglich, gezielt Datensätze zwischen verschiedenen Apps auszutauschen. Hierbei wird dem Entwickler freigestellt, ob die Daten allen (kompatiblen) Apps zur Verfügung gestellt werden sollen oder das Ziel explizit definiert wird. In letzterem Fall ist eine Absprache zwischen den Entwicklern beider Apps notwendig, um die Art des Transfers (z.B. exportierte Activity vs. Aktion) festzulegen. In jedem Fall wird garantiert, dass der Transfer nur zwischen Apps stattfinden kann, deren Intent-Filter entsprechende Kompatibilität definieren.

Komplexe Objekte können durch zwei Arten von Serialisierung übertragen werden. *Serializable* ist einfacher zu verwenden, da es keinerlei zusätzlichen Aufwand vom Entwickler erfordert; *Parclelable* erlaubt dagegen, das Vorgehen selbst festzulegen und damit auf die absolut nötigen Datenfelder zu reduzieren (siehe Abschnitt 4.2.1). Benchmarks zeigen, dass Parcelable damit deutlich weniger zeitlichen Overhead produziert [Bre13] (mehr dazu in Abschnitt 6.1); jedoch ist solch ein Overhead i.d.R. nur bei einer größeren Anzahl von Datensätzen relevant, beim Transfer einzelner Objekte fällt der entsprechende Zeitgewinn kaum ins Gewicht.

Obwohl der Intent-Ansatz den Transfer von Daten scheinbar auf eine bestimmte Ziel-App einschränkt (entweder direkt oder per Auswahl durch den Anwender), garantiert dies nicht, dass die Daten vor unerlaubten Zugriffen anderer Apps geschützt sind. Beispielsweise kann die Methode *getRecentTasks()* des *ActivityManagers* verwendet werden, um die Intents, die diese Tasks erzeugt hatten, inklusive angehängter Extras auszulesen. Eine App benötigt hierfür lediglich die Berechtigung *GET_TASKS* [CFGW11].

2.1.3 Content Provider

Jede Android-App hat ein eigenes internes Dateisystem, auf das nur die App selbst Zugriff hat. Private Daten einzelner Apps sind somit zunächst nicht von anderen Prozessen lesbar. Um die Freigabe bestimmter Daten an externe Apps zu ermöglichen, kann ein *Content Provider* [dev17d] implementiert werden; dieser definiert verschiedene Methoden (z.B. *insert()*, *update()*, *delete()*), die von externen Apps aufgerufen werden können.

Bekannte Beispiele sind unter anderem die Standard-Apps *Kalender* und *Kontakte*: Während deren Daten direkt verwaltet und angezeigt werden können, ist es ebenfalls möglich, von anderen Apps darauf zuzugreifen, sofern diese z.B. die Berechtigung *READ_CONTACTS* oder *READ_CALENDAR* haben. Abbildung 2.2 zeigt schematisch die Rolle eines Content Providers als Schnittstelle zwischen externen Apps und den gespeicherten Daten.

Da sämtliche Zugriffe über die vom Content Provider implementierten Methoden erfolgen, kann eine Verschlüsselungsstrategie genutzt werden, um gespeicherte Daten vor unerlaubten Direktzugriffen auf das Dateisystem (z.B. auf gerooteten Geräten) abzusichern, beispielsweise durch Verschlüsselung einer darunterliegenden Datenbank. Aziz zeigt beispielhaft, wie ein

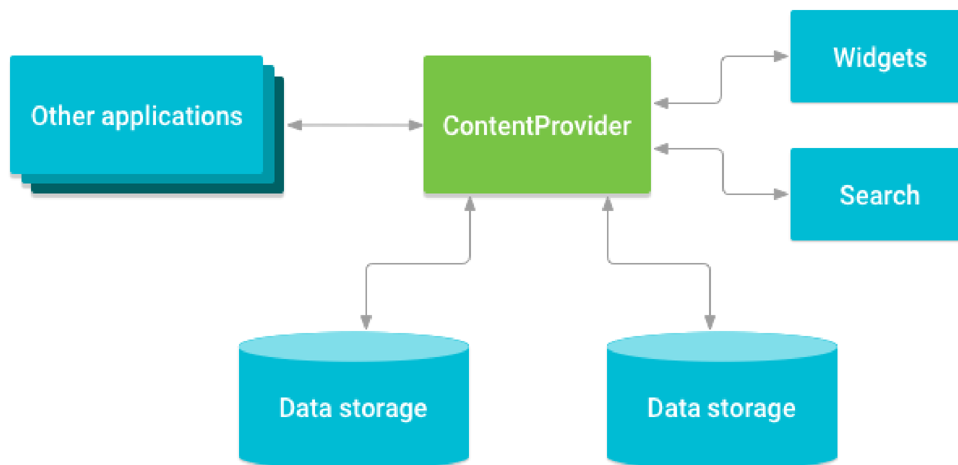


Abbildung 2.2: Beziehung zwischen Content Provider und anderen Komponenten [dev17d]

Content Provider mit SQLCipher geschützt werden kann [Azi13]. Im Gegensatz zu den zuvor beschriebenen Ansätzen muss hierbei auch kein (potentiell unsicherer) Austausch des Schlüssels zwischen Apps stattfinden, da dieser nur dem Provider selbst bekannt sein muss.

Berechtigungen

Jede App, die einen Content Provider bereitstellt, muss die Berechtigungen definieren, die andere Apps verwenden können. Hierbei gibt es folgende Möglichkeiten [dev17f]:

- **Provider-Level** - Dauerhafter Lese- und/oder Schreibzugriff auf alle verfügbaren Daten
- **Path-Level** - Dauerhafter Lese- und/oder Schreibzugriff auf bestimmte Pfade
- **URI** - Temporärer Zugriff auf bestimmte URIs (durch Mitsenden eines Flags)

Je nachdem, welche Zugriffsmodelle erlaubt sein sollen, können Berechtigungen mit beliebiger Granularität definiert werden. Jede App, die Zugriff auf Daten erhalten möchte, muss die entsprechende Berechtigung über ihr Manifest anfordern, wodurch dem Anwender bei der Installation mitgeteilt wird, auf welche Daten die jeweilige App nutzen möchte.

Bewertung

Mit Content Providern ist es möglich, Daten für externe Apps freizugeben, ohne sie vollständig öffentlich zu machen. Durch das Berechtigungssystem wird prinzipiell verhindert, dass unbefugte Apps Zugriff erhalten, da der Anwender den angeforderten Berechtigungen bei Installation einer App explizit zustimmen muss. Jedoch ist dadurch allein keine absolute Sicherheit der Daten gewährleistet.

Während Anwender früher verpflichtet wurden, alle angeforderten Berechtigungen einer App schon bei der Installation zu gewähren, ist dies seit Android 6.0 anders: Bei jeder Operation der App, die eine (noch nicht gewährte) Berechtigung benötigt, wird der Anwender zunächst um Erlaubnis gefragt und kann dabei ggf. die Berechtigung verweigern [dev17m]; bereits gewährte Berechtigungen können über die Einstellungen zurückgezogen werden. Dies setzt jedoch voraus, dass die entsprechende Berechtigung als *dangerous* deklariert wurde; andernfalls wird der Zugriff vom System automatisch gewährt [dev17l].

Diese Neuerung gilt jedoch nur, wenn die App entsprechend den aktuellen Richtlinien entwickelt wurde, da ältere Apps bei nicht gewährten Berechtigungen meist abstürzen. Ebenfalls ist es seitens des App-Entwicklers möglich, beliebige Funktionen zu unterdrücken, wenn bestimmte Berechtigungen nicht gewährt wurden. Generell ist es auch sehr wahrscheinlich, dass Anwender Berechtigungsanfragen einer App aus Bequemlichkeit bestätigen [BCG13].

Neben den Möglichkeiten für Entwickler, die Berechtigungssteuerung auf eine Weise zu nutzen, die Anwender zur manuellen Freigabe der Berechtigung manipuliert, können auch Sicherheitslücken im Android-System ausgenutzt werden. Ein Beispiel hierfür sind so genannte *Privilege Escalation Attacks*, bei denen eine unbefugte App durch Zugriff auf eine (möglicherweise nicht ausreichend geschützte) berechtigte App wiederum Zugriff auf die eigentlichen privaten Daten erhalten kann [DDSW10].

Generell reicht es bei der Zugriffssteuerung durch Berechtigungen daher nicht aus, den Content Provider selbst abzusichern; vielmehr müsste jede App, die entsprechende Berechtigungen verwendet, ebenfalls vor unerwünschten Zugriffen geschützt sein, was in der Praxis kaum durchsetzbar ist.

2.1.4 Clipboard

Analog zur Kopieren/Einfügen-Funktionalität traditioneller Betriebssysteme bietet auch Android die Möglichkeit, Daten über eine Zwischenablage [dev17e] direkt zwischen Apps zu übertragen. Objekte für den Android *ClipboardManager* können aktuell eines der folgenden Formate haben:

- **Text** - Übergabe der Daten in Form eines Strings
- **URI** - Übergabe der Referenz auf einen Datensatz
- **Intent** - Übergabe eines Intent-Objekts

Wichtig ist hierbei, dass URI und Intent nur auf Systemen ab API-Level 11 angeboten werden; bei älteren Versionen war nur das Ablegen von Text möglich. In der neuen Implementierung werden zusätzlich Metadaten zum jeweiligen Datensatz gespeichert, was einer App erlaubt, beim Zugriff auf die Zwischenablage lediglich kompatible Datensätze auszulesen. Ebenfalls wird eine Funktion angeboten, die URIs und Intents in Textform umwandelt, damit sie von der empfangenden App als solcher verarbeitet werden können.

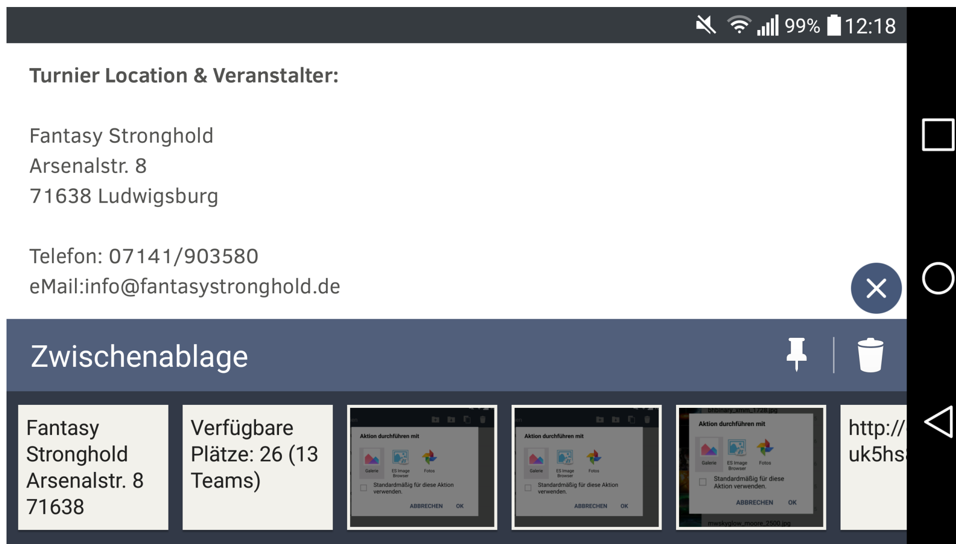


Abbildung 2.3: Implementierung der Zwischenablage mit mehreren Objekten

Um einen Datensatz zu teilen, muss dieser zunächst explizit von einer App in den *Clipboard-Manager* übertragen werden, erst danach können andere Apps darauf zugreifen. Wichtig ist hierbei, dass die originale Android-Zwischenablage nur ein einziges *ClipData*-Objekt enthalten kann; sobald ein weiteres Element dort abgelegt wird, verschwindet das vorhergehende. Manche Gerätehersteller installieren jedoch eigene Clipboard-Implementierungen auf ihren Geräten, die mehrere Objekte gleichzeitig erlauben (siehe Abbildung 2.3).

Bewertung

Mit der zweiten Generation des *ClipboardManagers* können neben textbasierten Daten und serialisierten Objekten auch Datensätze von Content Providern (via URIs) komfortabel übertragen werden. Dadurch ist es möglich, Daten an Apps weiterzugeben, die sonst keine Zugriffsberechtigung dafür haben. Dies setzt voraus, dass der jeweilige Content Provider temporäre Zugriffe auf die entsprechende URI erlaubt (siehe Abschnitt 2.1.3).

Ein großer Nachteil dieses Ansatzes ist jedoch die Sicherheit: Während z.B. Datensätze in Content Providern durch das Android-Berechtigungssystem zumindest prinzipiell vor unerlaubten Zugriffen geschützt werden können, ist die Zwischenablage für jede installierte App einsehbar und veränderbar, da für den Zugriff keinerlei spezielle Berechtigung erforderlich ist.

Einerseits kann daher nicht verhindert werden, dass jede App die dort abgelegten Daten ausliest; andererseits können Daten, die zwischen zwei Apps ausgetauscht werden sollen, jederzeit von Dritten manipuliert werden. Für den Transfer kritischer Daten müsste daher eine zusätzliche Sicherheitskomponente (z.B. Verschlüsselung) verwendet werden, welche wiederum eine Absprache zwischen den beteiligten Apps erfordert.

2.2 Alternative Ansätze

Die im Folgenden beschriebenen Verfahren sollen Verbesserungen gegenüber traditionellen Android-Komponenten bieten. Der Fokus liegt dabei wahlweise auf Sicherheit, feingranularen Berechtigungen und/oder Bedienbarkeit.

2.2.1 MetaService

Als Alternative bzw. Optimierung gegenüber der existierenden Zwischenablage wurde der sogenannte *MetaService* [CBJP11] konzipiert. Fokus liegt hierbei darauf, den Austausch von komplexen Objekten möglichst einfach zu gestalten.

Während textbasierte Datensätze über das Android-Clipboard direkt übertragen werden können, ist für den Transfer von Objektstrukturen (ohne Umweg über Serialisierung) die Weitergabe per URI und damit die Abwicklung über einen Content Provider notwendig. Dies erfordert zusätzlichen Aufwand seitens des Entwicklers, da entsprechende Zugriffsmethoden auf App-Seite implementiert werden müssen.

MetaService stellt hierbei ein Konzept vor, welches die entsprechende Funktionalität in einer vom Service zur Verfügung gestellten Manager-Klasse kapselt (siehe Abbildung 2.4). Apps müssen dadurch lediglich einer Instanz des *MSManagers* den gewünschten Datensatz mit *setObject()* zuweisen, eine andere App kann dieses dann wiederum mit *getObject()* auslesen.

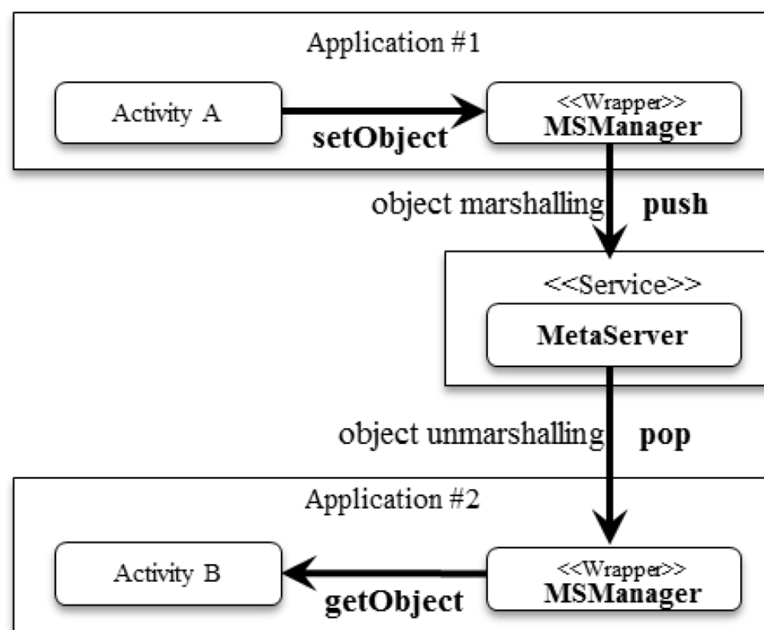


Abbildung 2.4: Grundlegende Architektur des MetaService [CBJP11]

Dieses Prinzip wurde analog zur ersten Generation des Android-Clipboards konzipiert, bei der dieser einfache Zugriff für textbasierte Datensätze vorhanden war. Analog zur nativen Clipboard-Implementierung existiert auch hier ein Objekt nur so lange in der Zwischenablage, bis es durch ein neues ersetzt wird. Ob das Objekt dabei persistent gespeichert wird oder nur im RAM liegt, ist nicht spezifiziert.

Die Kernkomponente (analog zum Android Clipboard) stellt hierbei der sogenannte *MetaServer* dar, welcher als Service implementiert wird. Hierbei gibt es zwei Varianten:

- **System Service** - Der Service wird als neue Systemkomponente Teil des Android Application Frameworks. Als solcher kann er von jeder App direkt über den globalen *ServiceManager* aufgerufen werden. Von Nachteil ist hierbei, dass Änderungen am Android-Betriebssystem nur auf gerooteten Geräten und nur mit größerem Aufwand möglich sind, weshalb dies für „normale“ Anwender keinen sinnvollen Ansatz darstellt.
- **Remote Service** - Die App, die den Service bereitstellt, wird regulär installiert. Eine aufrufende App startet den Service über einen Intent und erzeugt ein Binding als Basis für den Datenaustausch. Im vorliegenden Fall stellt *MSManager* eine entsprechende Methode bereit. Der Nachteil hierbei ist, dass der Anwender in der Lage ist, die entsprechende Service-App zu deinstallieren (was die Funktionalität aller kompatiblen Apps stark einschränken würde), ebenfalls kann Manipulation (z.B. durch Installation einer App mit identischem Namen) nicht ausgeschlossen werden.

MetaService unterscheidet beim Transfer zwischen „serialisierbaren“ (z.B. mit Implementierung von *java.io.Serializable*) und „nicht serialisierbaren“ Objekten. Letztere werden stattdessen mit einem alternativen Ansatz für die Übertragung umgewandelt, die Vorgehensweise wird ist allerdings nicht konkret definiert.

Um beide Varianten zu unterschieden, wird ein entsprechender Parameter mitgeschickt. Sämtliche Operationen zur Umwandlung (marshalling/unmarshalling) und Übertragung der Daten werden automatisch innerhalb der *MSManager*-Komponente abgewickelt, so dass dem Entwickler keinerlei zusätzlicher Aufwand entsteht.

Bewertung

Im Gegensatz zum nativen Android-Clipboard verzichtet MetaService vollständig auf die Verwendung von Content Providern zum Austausch komplexer Objekte. Daten können daher nicht per Referenz (URI) übertragen werden, sondern nur durch Ablegen des vollständigen Objekts auf dem *MetaServer*. Einfache Bedienbarkeit für den Entwickler wird hierbei priorisiert, es muss (und kann) kein manueller Eingriff in das Verfahren stattfinden.

In Bezug auf Sicherheit hat der Ansatz jedoch dieselben Nachteile wie das native Android-Clipboard: Zugriff auf den Service und damit die abgelegten Daten ist für jede installierte App auf dem Gerät möglich, daher ist auch dieses Konzept für dem Austausch kritischer bzw. privater Daten nicht empfehlenswert.

2.2.2 Porscha

Porscha [OBM10] stellt einen Ansatz für die durchgehende Sicherstellung von DRM-Schutz für verschiedenste Arten von Datenquellen vor. Hierbei wird nicht nur der Schutz von Daten innerhalb des Mobilgeräts betrachtet, sondern auch (sofern zutreffend) der Weg zwischen Sender und Empfänger (z.B. bei SMS oder eMails).

DRM steht für *Digital Rights Management* [SY06], dabei handelt es sich allgemein um Verfahren, die den Zugriff auf digitale Güter kontrollieren sollen. Bekannt sind sie vor allem durch den Verkauf elektronischer Medien wie z.B. Musik, Videos und eBooks. Während die unautorisierte Weitergabe der Dateien selbst nicht verhindert werden kann, wird sichergestellt, dass diese nur vom tatsächlichen Käufer geöffnet bzw. abgespielt werden können. Allgemein bedeutet es, dass die Richtlinien, nach denen Zugriffe auf Datensätze stattfinden dürfen, innerhalb des Datensatzes selbst definiert sind und daher nicht umgangen werden können.

Das Porscha-System basiert auf zwei grundlegenden Komponenten: Verwendung eines asymmetrischen Verschlüsselungsverfahrens zur Datenübertragung von externen Quellen und Kontrolle der Datenzugriffe auf dem Gerät über einen zusätzlichen *Mediator*. Im Folgenden werden die einzelnen Funktionen dieser Komponenten am Beispiel von SMS-Empfang erläutert, das Prinzip ist jedoch auf alle Arten von Datenzugriffen anwendbar.

Ähnlich wie beim Kauf digitaler Medien muss auch hier sichergestellt werden, dass nur der beabsichtigte Empfänger die Nachricht lesen kann. Das mobile Netzwerk wird als unsicheres Medium betrachtet, so dass der Schutz bereits beim Versand der Nachricht beginnen muss.

Dabei wird *IBE (Identity Based Encryption)* [CS11] genutzt, um die Nachricht zu verschlüsseln. Der *Public Key* wird von der (eindeutigen) Telefonnummer des Empfängers abgeleitet, wobei nur der Inhaber der Telefonnummer den zugehörigen *Private Key* kennt. Letzteres könnte beispielsweise durch die Codierung auf der entsprechenden SIM-Karte oder durch Erhalt des Schlüssels bei Abschluss eines Mobilfunkvertrags sichergestellt werden.

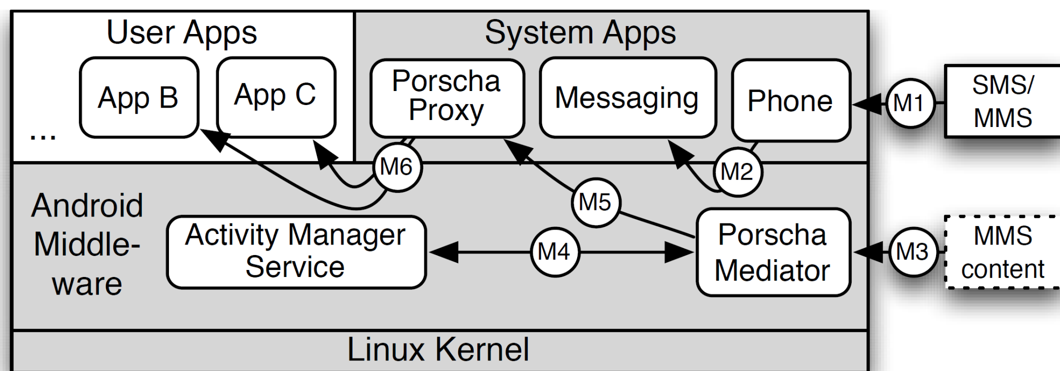


Abbildung 2.5: Integration des Porscha-Mediators beim SMS-Empfang [OBM10]

Sobald eine Nachricht auf dem Empfängergerät eintrifft (siehe Abbildung 2.5, M1), werden normalerweise alle Apps benachrichtigt, die einen Broadcast-Receiver für SMS-Empfang definieren (sofern diese auch die *RECEIVE_SMS* Berechtigung besitzen). Porscha unterbindet diese Benachrichtigung zunächst, lädt die Nachricht jedoch trotzdem in die Nachrichten-App des Systems herunter (M2).

Die Nachricht wird dann vom *Mediator* ausgewertet (M3), der in Zusammenarbeit mit dem *Activity Manager Service* auf Basis der eingebetteten DRM-Richtlinie überprüft, welchen Apps tatsächlich Zugriff gestattet werden soll (M4). Nur diesen Apps wird dann über den *Porscha Proxy* eine entsprechende Benachrichtung geschickt (M5 und M6); der Proxy ist dabei nur deshalb notwendig, weil Broadcasts nicht aus der Middleware-Ebene gesendet werden können.

Auch andere Systeme für Datenfreigaben auf der Android-Plattform werden bei Porscha entsprechend erweitert. So schickt beispielsweise jeder Intent eine Zugriffsrichtlinie mit, genauso werden diese zusammen mit einzelnen Datensätzen in Content Providern gespeichert, wobei bei jedem Aufruf der Mediator zur Auswertung zwischengeschaltet ist.

Bewertung

Das Porscha-Konzept bietet eine Verfeinerung des bestehenden Berechtigungsmanagements durch den Einsatz von Datenobjekt-spezifischen Zugriffsrichtlinien. Von externen Quellen empfangene Daten werden darüber hinaus für die Übertragung verschlüsselt, damit beim Transfer über ein unsicheres Medium unberechtigte Zugriffe verhindert werden.

Der Vorteil dieses Ansatzes ist die Möglichkeit, für jeden Datensatz aus beliebiger Quelle exakte Zugriffsberechtigungen zu definieren; neben einer Einschränkung der Zugriffserlaubnis auf bestimmte Apps könnten hierbei auch weitere Kontextinformationen (etwa Uhrzeit, Standort, etc.) berücksichtigt werden. Darüber hinaus darf aufgrund des DRM-Prinzips nur der Urheber eines Datensatzes dessen Richtlinien festlegen, so dass eine App nicht eigenmächtig Zugriff erlangen kann.

Ein großer Nachteil ist jedoch die Art der Implementierung: Für den Einsatz von Porscha müssen an vielen verschiedenen Stellen des Android-Betriebssystems Änderungen vorgenommen werden, sowohl auf Ebene der System-Apps als auch innerhalb der Middleware. Dies erfordert die Installation eines entsprechenden alternativen Android-Builds, was für „normale“ Anwender kaum realisierbar ist.

Ein weiteres Problem besteht bei der Kompatibilität, da dem Sender einer SMS beispielsweise nicht bekannt ist, ob der Empfänger ein Porscha-konformes System nutzt. Um dies zu umgehen, müsste eine unverschlüsselte Version der Nachricht als Fallback für traditionelle Systeme mitgeschickt werden, wodurch die gewonnene Sicherheit verloren ginge.

Insgesamt wäre ein solches System zwar hervorragend dafür geeignet, die Sicherheit von Datenübertragungen innerhalb und außerhalb von Mobilgeräten zu gewährleisten, jedoch ist es nur praktikabel, wenn es von allen Geräten unterstützt wird.

2.2.3 Kynoid

Kynoid [SKS13] bietet eine Plattform für die Definition und Durchsetzung feingranularer Zugriffsrichtlinien für einzelne Datensätze. Das System basiert auf TaintDroid [EGH+14], einem Analyse-Tool, welches Zugriffe auf geschützte Informationen in Echtzeit kontrolliert.

Dabei werden den von Datenquellen produzierten Informationen zusätzliche versteckte Werte *(„Taints“)* hinzugefügt. Bei Verarbeitung von Daten durch einen sekundären Empfänger werden diese Markierungen analysiert, dadurch kann erkannt werden, wenn Apps aus geschützten Quellen stammende Daten weitergeben.

Kynoid erweitert dieses System durch einen *Policy Manager*, über den der Anwender feingranulare Zugriffsrichtlinien für Datenquellen definieren kann. Im Folgenden wird beschrieben, wie Zugriffsverläufe auf Basis geschützter Daten beobachtet und ausgewertet werden.

Apps fordern den Zugriff auf einen Datensatz beispielsweise von einem Content Provider (siehe Abbildung 2.6, 1) an. Dieser überprüft über den *Kynoid System Service* die dazugehörigen Richtlinien (2); dabei wird eine neue Referenz mit eindeutiger ID im System angelegt (3).

Jede ID entspricht einer Variablen, die Informationen aus geschützten Quellen enthält. Werden im Laufe der Programmausführung weitere Variablen auf deren Basis erzeugt, erhalten diese ebenfalls eine ID; darüber hinaus werden dazugehörige Richtlinien über einen Abhängigkeitsgraph miteinander verknüpft (4).

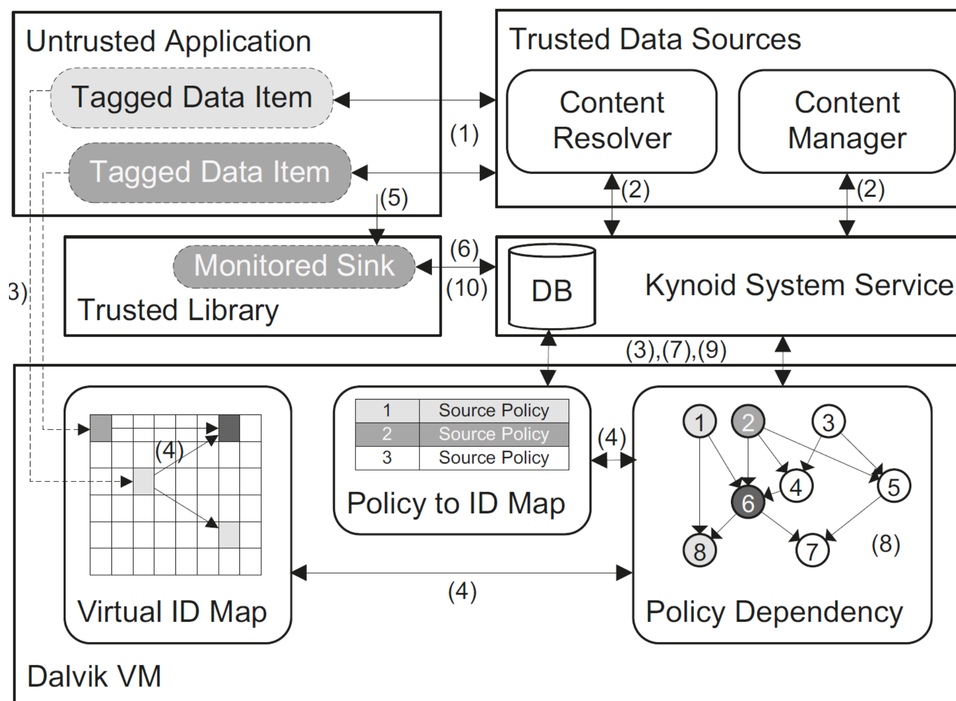


Abbildung 2.6: Architektur des Kynoid-Frameworks [SKS13]

Sobald ein Datensatz bei einem Empfänger ankommt (5), werden wiederum über den *Kynoid System Service* (6,7) alle Abhängigkeiten im Graph zurückverfolgt (8) und die insgesamt daraus resultierende Sicherheitsrichtlinie berechnet (9). Diese Berechnung könnte zwar auch schrittweise bei der Erstellung neuer Knoten im Graph durchgeführt werden; jedoch besteht die Möglichkeit, dass sich während der Ausführung Richtlinien ändern, so dass nur eine Berechnung am Ende diese mit berücksichtigen kann. Die auf Basis der Sicherheitsrichtlinie getroffene Entscheidung wird schließlich an den Empfänger zurückgegeben.

Insgesamt werden über Kynoid demnach zwei separate Zugriffsmodelle geprüft: Beim Direktzugriff auf eine Datenquelle werden entsprechende Richtlinien *(Source Policies)* ausgewertet, bei Widerspruch kann die entsprechende Anfrage sofort abgelehnt werden. Darüber hinaus wird die Weitergabe geschützter Daten zwischen Prozessen verfolgt; hierbei wird die Operation beim endgültigen Empfänger verweigert, wenn der Gesamtverlauf an einer oder mehreren Stellen definierten Richtlinien widerspricht.

Bewertung

Anders als alle anderen hier vorgestellten Ansätze bietet Kynoid nicht nur eine Zugriffskontrolle für die Datenquellen selbst, sondern kann auch weiterverfolgen, inwiefern Daten von berechtigten Apps an Dritte übermittelt werden. Bei Porscha wird beispielsweise zwar verhindert, dass unberechtigte Apps auf geschützte Daten zugreifen können, jedoch ist eine berechtigte App trotzdem in der Lage, die entschlüsselten Informationen zu kopieren und weiterzugeben.

Insgesamt hat Kynoid jedoch denselben Nachteil wie Porscha: Für die Implementierung sind tiefgreifende Änderung am Betriebssystem notwendig, was die Installation eines alternativen Android-Builds voraussetzt. Im Gegensatz zu Porscha können Anwender diese Entscheidung jedoch individuell treffen, da keine externen Abhängigkeiten bestehen.

2.2.4 Secure Data Container

Um eine Plattform für den sicheren Austausch von Daten bereitzustellen, deren Zugriffe nicht über das traditionelle Android-Berechtigungsmanagement gesteuert werden, wurde der *Secure Data Container (SDC)* [SM15; SM16] entworfen. Hierbei handelt es sich um eine Art alternativen Content Provider, der Daten für andere Apps in einer internen Datenbank speichert und Freigaben verwaltet. Der SDC wurde als Bestandteil der *Privacy Management Platform (PMP)* [SM13; SM14] entwickelt (mehr dazu in Abschnitt 5.1).

Die Zuordnung, welche Daten von welchen Apps gelesen werden können, wird ausschließlich vom Besitzer eines Datensatzes über den Container selbst festgelegt; eine Berechtigung kann daher nicht von externen Apps angefordert werden.

Eine App, die den Container verwenden möchte, muss sich zunächst bei diesem registrieren und bekommt eine eindeutige ID zugewiesen. Gespeicherte Datensätze der App werden unter deren ID und einem eindeutigen Schlüssel abgelegt und dürfen zunächst nur von dieser App gelesen werden.

Jeder Datensatz kann mit individuellen Apps geteilt werden, indem sein *shareable*-Flag gesetzt und ein entsprechender Eintrag für die jeweils berechtigte App in der Freigabe-Tabelle hinterlegt wird. Abbildung 2.7 zeigt ein generisches Schema für die Implementierung auf Basis einer relationalen Datenbank.

Das Schema des SDC kann darüber hinaus erweitert werden, um Objekte zu speichern, die aus mehreren Feldern bestehen: Hierbei wird für jedes zu speichernde Feld im Originalobjekt ein entsprechendes Feld in der Datenbank definiert (die Abbildung primitiver Datentypen kann hierbei analog geschehen, sofern die Datenbank-Implementierung diese unterstützt).

SQLite-Datenbanken werden standardmäßig in Klartext im Dateisystem der App abgelegt und sind damit anfällig für unberechtigte Zugriffe über das Dateisystem. Um dies zu verhindern, wird beim SDC die gesamte Datenbank verschlüsselt, wobei nur der Container selbst den entsprechenden Schlüssel kennt.

Zugriffe durch externe Apps müssen daher über die vom Container bereitgestellten Methoden abgewickelt werden, der auf Basis der hinterlegten Freigaben entscheidet, ob einer Anfrage stattgegeben wird. Schreibzugriff wird aus Sicherheitsgründen jedoch nur dem Urheber eines Datensatzes gewährt. Um die Datenbank endgültig unleserlich zu machen, verfügt der Container außerdem über die Möglichkeit, den eigenen Schlüssel zu löschen.

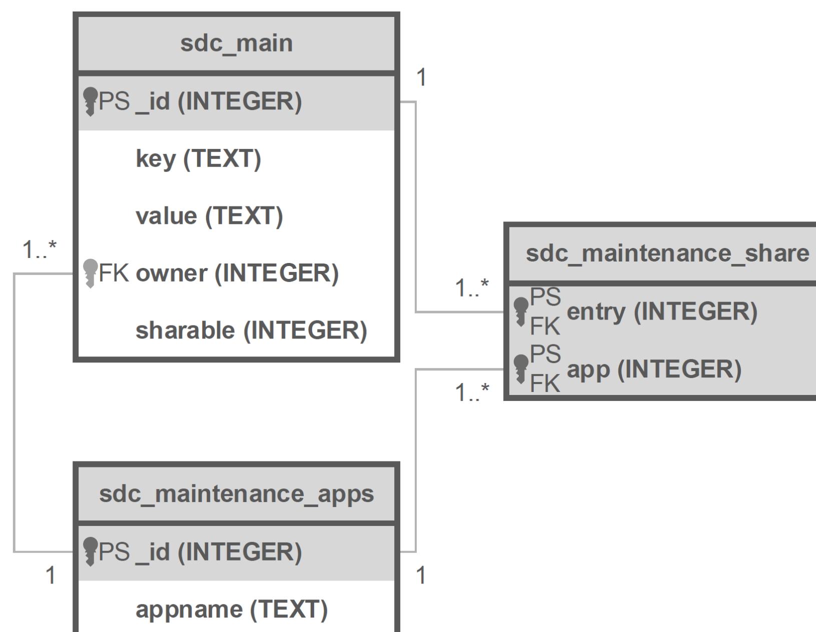


Abbildung 2.7: Relationales Schema für die Freigabeverwaltung des SDC [SM15]

Bewertung

Im Gegensatz zum traditionellen Content Provider stellt dieser Ansatz eine deutliche Verbesserung hinsichtlich der Sicherheit der gespeicherten Daten dar: Dadurch, dass Freigabeberechtigungen explizit innerhalb des Containers definiert werden müssen, kann eine App nicht eigenmächtig Zugriff auf Datensätze erlangen. Durch die Verschlüsselung wird außerdem garantiert, dass diese Sicherheit nicht durch Umgehung der vom Container bereitgestellten Zugriffsmethoden ausgehebelt werden kann.

Bei einer objektbasierten Implementierung ist jedoch erforderlich, dass für jede unterstützte Objektklasse eine eigene Tabelle innerhalb der Datenbank des SDC angelegt wird, um deren individuelle Strukturen abzubilden. Auch bedeutet dies einen hohen Aufwand beim Speichern und Auslesen von Datensätzen, da jedes Objekt zunächst in einzelne Felder „zerlegt“ und bei Rückgabe wieder „zusammengesetzt“ werden muss.

Generell ist von Seiten des Containers immer vollständige Kenntnis über die Zusammensetzung des jeweiligen Objekts nötig. Die Änderung einer bestehenden Klassenstruktur sowie das Hinzufügen neuer Klassen ist daher immer mit großem Änderungsaufwand seitens der Container-Implementierung verbunden.

2.3 Zusammenfassung

Die folgende Tabelle fasst zusammen, welche der in Abschnitt 1.3 definierten Anforderungen von den in diesem Kapitel vorgestellten Ansätzen jeweils erfüllt werden. Durchgestrichene Eigenschaften werden dabei lediglich durch das Android-Berechtigungssystem durchgesetzt und bieten keine tatsächlichen Sicherheitsgarantien, da unberechtigte Zugriffe (beispielsweise über *Privilege Escalation Attacks*, siehe Abschnitt 1.1) nicht ausgeschlossen werden können.

	C	**I**	**A**	**F**	**G**	**U**
Dateisystem	~~Ja~~	~~Ja~~	Ja	Nein	Ja	Ja
Intent	Nein	Ja	Nein	Ja	Ja	Ja
Content Provider	~~Ja~~	~~Ja~~	Ja	Ja	Nein	Ja
Clipboard	Nein	Nein	Nein	Nein	Ja	Ja
MetaService	Nein	Nein	Nein	Nein	Ja	Ja
Porscha	Ja	Ja	Ja	Ja	Ja	Nein
Kynoid	Ja	Ja	Ja	Ja	Ja	Nein
Secure Data Container	Ja	Ja	Ja	Ja	Nein	Ja

Tabelle 2.1: Vergleich der betrachteten Ansätze hinsichtlich der Anforderungen

[C]onfidentiality (Vertraulichkeit) - Von den nativen Android-Funktionen bieten lediglich Content Provider die Möglichkeit, individuelle Zugriffsrechte für andere Apps zu definieren. Beim Dateisystem kann dieses Recht dagegen nur pauschal vergeben werden. Jedoch wird in beiden Fällen das Android-Berechtigungssystem genutzt, welches keinen ausreichenden Schutz vor unerlaubten Zugriffen bietet. Bei den alternativen Ansätzen werden Zugriffskontrollen durch zusätzliche Sicherheitsrichtlinien durchgesetzt; einzige Ausnahme ist MetaService, bei dem keinerlei Zugriffsbeschränkungen existieren.

[I]ntegrity (Unversehrtheit) - Ähnlich wie bei der vorhergehende Eigenschaft kann auch der Schutz vor unberechtigten Änderungen bei nativen Komponenten nicht garantiert werden. Einzige Ausnahme sind Intents, da diese nicht auf einem Datenspeicher basieren, sondern einmalig erzeugt werden. Apps können diese zwar einsehen, jedoch nicht verändern. Clipboard und MetaService sind am problematischsten, da deren Inhalt von Apps beliebig verändert werden kann. Alle weiteren alternativen Ansätze bieten dieselben Garantien wie für Lesezugriffe.

[A]vailability (Verfügbarkeit) - Hierbei muss zwischen Verfügbarkeit von Plattform und Daten unterschieden werden. Als Service implementierte Verfahren sind dauerhaft verfügbar; Intents werden dagegen einmalig gesendet, wobei der definierte Empfänger möglicherweise nicht erreichbar ist. Die Verfügbarkeit der Daten ist dagegen nur bei Verfahren garantiert, die diese persistent speichern. Clipboard und MetaService sind jedoch von vornherein als flüchtige Speicherbereiche konzipiert, die jederzeit überschrieben werden können.

[F]eingranulare Berechtigungen - Das Android-Dateisystem kann nur pauschale Zugriffsrechte gewähren; Clipboard und MetaService sind für alle Apps gleichermaßen verfügbar. Content Provider erlauben, Berechtigungen freingranular anzubieten, jedoch wird von dieser Option eher selten Gebrauch gemacht; insbesondere kann dies nur durch den Entwickler definiert werden, nicht aber durch den Anwender. Porscha, Kynoid und SDC ermöglichen dagegen, Zugriffsberechtigungen für einzelne Datensätze individuell festzulegen.

[G]enericity (Allgemeine Anwendbarkeit) - Sowohl im Dateisystem als auch über Intents und Clipboard können beliebige Datenstrukturen abgelegt werden (Objekte jedoch nur in serialisierter Form); MetaService ermöglicht dagegen das Ablegen beliebiger Objekte. Content Provider und SDC erlauben nur die Verarbeitung bestimmter unterstützter Objektstrukturen. Porscha und Kynoid ergänzen lediglich die Kontrolle der Zugriffsberechtigungen für bestehende Ansätze und bieten keine alternativen Transfermethoden.

[U]sability (Benutzerfreundlichkeit) - Die Verwendung nativer Komponenten ist für Entwickler sehr einfach, da vom System passende Schnittstellen zur Verfügung gestellt werden. Der Austausch von Daten über die Zwischenablage und analog dazu über den MetaService ist dabei am komfortabelsten. Porscha und Kynoid erfordern die Installation eines alternativen Android-Systems und können daher auf traditionellen Geräten nicht genutzt werden.

Insgesamt ist klar erkennbar, dass keines der in diesem Kapitel vorgestellten Systeme alle Anforderungen erfüllt. Lediglich alternative Android-Builds bieten ausreichende Sicherheit für Daten, stellen jedoch keinen für die Allgemeinheit praktikablen Ansatz dar.

3 Der Secure Object Container

Das folgende Kapitel beschreibt den konzeptionellen Aufbau einer Plattform, über die beliebige Datenobjekte App-übergreifend bereitgestellt werden können. Ziel ist, alle in Abschnitt 1.3 definierten Anforderungen zu erfüllen. Betrachtet man den in Abschnitt 2.3 zusammengefassten Vergleich bestehender Ansätze, wird deutlich, dass der *Secure Data Container (SDC)* bereits einen Großteil der Anforderungen abdeckt, jedoch nur die Ablage bestimmter vordefinierter Klassenstrukturen ermöglicht. Für eine Austauschplattform, die von allen Apps und damit für beliebige Datenstrukturen nutzbar sein soll (siehe Abschnitt 1.3), ist dies jedoch nicht ausreichend. Im Gegensatz dazu erlaubt *MetaService* den Transfer aller Objekte, vernachlässigt dabei jedoch jegliche Sicherheitsaspekte.

Es ist daher naheliegend, eine Alternativvariante des SDC zu entwickeln, die analog zum MetaService die Ablage beliebiger Objektstrukturen ermöglicht. Da der MetaService darüber hinaus komfortablere Schnittstellen für App-Entwickler bietet, findet dieser Ansatz hier ebenfalls Anwendung. Die daraus resultierende Plattform wird im Folgenden als „Secure Object Container" (kurz: SOC) bezeichnet.

3.1 Schnittstellen

Um den grundlegenden Aufbau des SOC zu beschreiben, müssen zunächst die externen Schnittstellen definiert werden, über die Apps mit dem Container interagieren. Da es sich dabei in erster Linie um einen Datenspeicher handelt, müssen sämtliche CRUD-Operationen *(Create, Read, Update, Delete)* unterstützt werden. Darüber hinaus müssen Datensätze mit anderen Apps geteilt werden und Freigaben wieder rückgängig gemacht werden können:

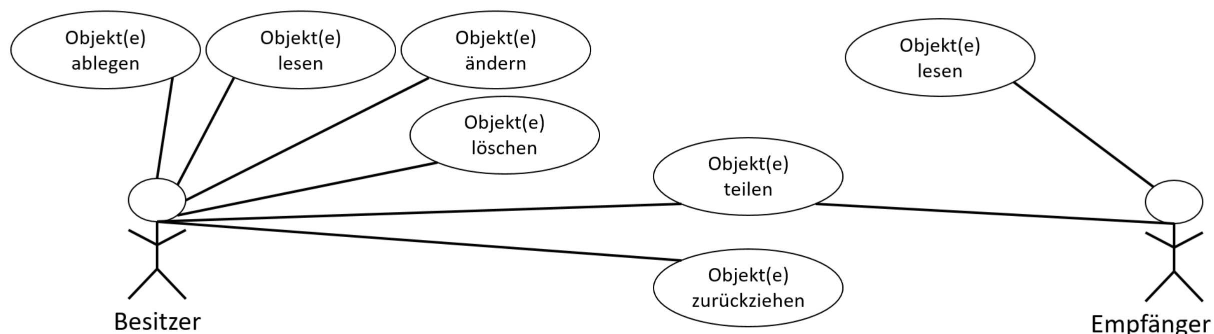


Abbildung 3.1: SOC-Datenoperationen aus Sicht von Besitzer und Empfänger

Da beliebige Arten von Objekten unterstützt werden sollen, dürfen keinerlei Anforderungen an deren interne Klassenstruktur gestellt werden. Damit ein Objekt innerhalb des SOC identifiziert werden kann, muss es daher unter einem eindeutigen Namen abgelegt werden. Dieser Name muss dabei App-übergreifend einzigartig sein, da potentiell jede App die Datensätze aller anderen Apps empfangen bekommen könnte.

3.1.1 Datenverwaltung

Um konkrete Datensätze anzusprechen, muss daher immer deren Name verwendet werden. Die Operationen *Create* und *Update* können dabei in einer Methode vereinigt werden:

- **putObject (name, object)** - Legt ein neues Objekt an, falls der Name noch nicht existiert. Ansonsten wird überprüft, ob die aufrufende App dessen Eigentümer ist, in diesem Fall wird die Änderung erlaubt. Analog zum SDC werden keine Änderungen an Fremdobjekten zugelassen.
- **getObject (name)** - Liest das Objekt, das unter dem angegebenen Namen abgelegt ist, sofern die aufrufende App dessen Eigentümer ist oder eine Freigabe erhalten hat.
- **deleteObject (name)** - Löscht das Objekt, sofern die aufrufende App dessen Eigentümer ist. Die Löschung eines durch Freigabe erhaltenen Objekts könnte hierbei das Löschen der entsprechenden Freigabe bewirken.

Für die Freigabeverwaltung muss zusätzlich der Name der App übermittelt werden, der die Zugriffsrechte gewährt bzw. entzogen werden sollen. Dabei muss für jede App ein eindeutiger Bezeichner existieren, der bei Abfrage von Datensätzen nicht vorgetäuscht werden kann.

- **shareObject (name, app)** - Erteilt der angegeben App Leserechte für ein Objekt, sofern die aufrufende App dessen Besitzer ist.
- **unshareObject (name, app)** - Zieht die Freigabe eines Objekts für eine bestimmte App zurück, sofern die aufrufende App dessen Besitzer ist.

Alle zuvor genannten Methoden sprechen nur einzelne Datensätze an; analog wäre aber auch möglich, Gruppen von Namen, Objekten oder Apps zu übergeben.

In jedem Fall muss eine App zunächst in der Lage sein, Kenntnis über den aktuellen (für sie sichtbaren) Datenbestand des SOC zu erlangen. Hierzu könnten Abfragen mit verschiedenen Arten von Filtern (oder Kombinationen davon) definiert werden, die die Namen der passenden Objekte zurückgeben:

- Eigene Objekte / durch Freigabe empfangene Objekte
- Freigaben von einer / an eine bestimmte App
- Objekte einer bestimmten Klasse

3.1.2 Kill Switch

Der Secure Data Container bietet die Möglichkeit, gespeicherte Daten mit einem „Kill Switch“ dauerhaft unleserlich zu machen (siehe Abschnitt 2.2.4), um im Fall einer Kompromittierung des Geräts sensible Inhalte vor unberechtigten Zugriffen zu schützen. Während dieses Ziel damit zweifellos erreicht wird, sorgt es für den Verlust *aller* abgelegten Daten.

Alternativ dazu sollte es für Apps auch eine Möglichkeit geben, ihre eigenen Daten und Freigaben zu löschen, ohne die Daten anderer Apps zu beeinträchtigen. Dies kann zwar manuell mit den zuvor beschriebenen Operationen für individuelle Datensätze durchgeführt werden, jedoch besteht dabei das Risiko, dass einzelne Elemente übrigbleiben. Eine Komplettlöschung säubert dagegen garantiert alle Datensätze sowie dazugehörige Freigaben einer App, was aus Sicht des App-Entwicklers deutlich komfortabler ist.

Darüber hinaus kann erwünscht sein, dass Datensätze einer App nach deren Deinstallation nicht länger verfügbar sind (analog zum Dateisystem einer App, welches bei Deinstallation ebenfalls gelöscht wird). Dabei könnte die entsprechende *cleanup*-Methode vom SOC automatisch ausgelöst werden, sobald eine App deinstalliert wurde.

3.1.3 API-Wrapper

Analog zum MetaService (siehe Abschnitt 2.2.1) stellt auch der SOC eine Wrapper-Komponente bereit, die von beliebigen Apps eingebunden werden kann. Dies ermöglicht Entwicklern, die öffentlichen Schnittstellen des SOC direkt anzusprechen, ohne dass zusätzlicher Implementierungsaufwand für die Verbindungsherstellung mit dem SOC oder die Umwandlung der Objekte für die Übertragung anfallen. Abbildung 3.2 zeigt schematisch die Kommunikation zwischen App, Wrapper und SOC:

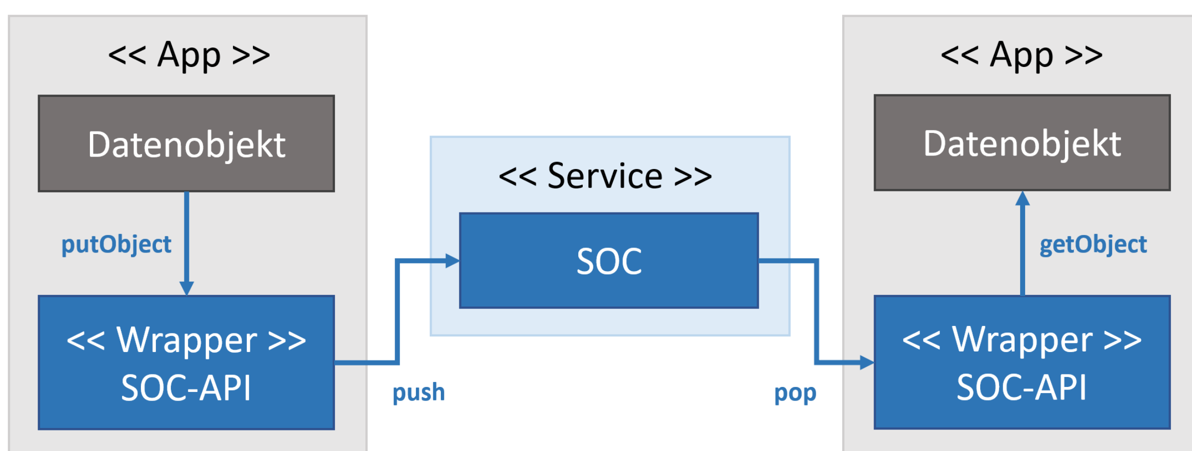


Abbildung 3.2: Kommunikation von Apps mit dem SOC über einen API-Wrapper

3.2 Schema

Damit mit den zuvor vorgestellten Methoden schnell und unkompliziert auf gespeicherte Datensätze und Freigabeberechtigungen zugegriffen werden kann, müssen alle relevanten Verwaltungsinformationen strukturiert abgespeichert werden. Angelehnt an die in Abschnitt 2.2.4 vorgestellte Tabellenstruktur des SDC ergeben sich für den SOC die im Folgenden vorgestellten mindestens benötigten Felder.

Damit Eigentümer und Freigabeziele von Objekten eindeutig zugewiesen werden können, müssen alle entsprechenden Apps unter ihrem eindeutigen Bezeichner beim SOC registriert sein. Da der SOC beliebige Objekte speichert, wird ein generisches Schema verwendet, bei dem ein einziges Feld für die Ablage der Objektdaten zur Verfügung steht. Dessen Metadaten beinhalten neben dem Speichernamen des Objekts und einer Referenz auf den Eigentümer lediglich den Namen der dazugehörigen Objektklasse. Freigaben werden durch eine Kombination aus Objekt-Name und App-Bezeichner des Empfängers definiert.

Abbildung 3.3 veranschaulicht die Beziehungen zwischen den einzelnen Feldern in Form einer relationalen Datenbank (diese Implementierung ist jedoch nur als Beispiel zu verstehen, weitere Varianten werden in Abschnitt 4.3 diskutiert). Primärschlüssel (PK) und Fremdschlüssel (FK) sind hierbei entsprechend gekennzeichnet.

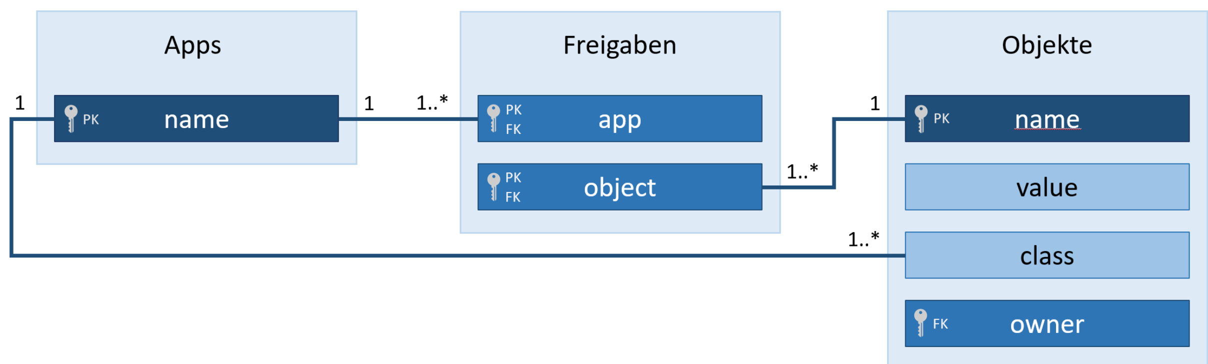


Abbildung 3.3: Relationales Datenbankschema des SOC

Im Gegensatz zum SDC ist es bei diesem Ansatz nicht möglich, Abfragen auf Basis einzelner Felder innerhalb eines Objekts zu stellen. Je nach verwendetem Verfahren sind dessen Daten in serialisierter Form nur begrenzt lesbar (mehr dazu in Abschnitt 4.2).

Bei textbasierten Serialisierungsverfahren wäre es zwar theoretisch denkbar, den Objektinhalt auf bestimmte Werte abzugleichen, jedoch würden solche Abfragen eine schlechtere Performance aufweisen, als es bei Abbildung einzelner Objektfelder auf ein relationales Schema der Fall wäre. Da die Objektdaten darüber hinaus verschlüsselt gespeichert werden sollen (siehe Abschnitt 3.3), ist diese Idee hier nicht sinnvoll anwendbar.

3.3 Verschlüsselung

Der in Abschnitt 2.2.4 vorgestellte SDC verschlüsselt die gesamte darunterliegende Datenbank, der Schlüssel ist dabei nur dem Container bekannt. Sobald eine App Zugriff auf einen Datensatz anfordert, wird zunächst die Datenbank entschlüsselt und daraufhin auf entsprechende Berechtigungen überprüft. Während dieses Vorgehen zweifellos die gespeicherten Daten vor unberechtigten Zugriffen schützt, hat es folgende grundlegende Nachteile:

- Auch wenn nur ein einziger Datensatz von der Anfrage betroffen ist, muss die vollständige Datenbank entschlüsselt und danach wieder verschlüsselt werden
- Selbst bei eigentlich unberechtigten Zugriffsanfragen ist eine Entschlüsselung notwendig

Beide Situationen erzeugen einen deutlichen Overhead bei Zugriffen auf die Datenbank. Verbessern lässt sich dies mit der Erkenntnis, dass die Metadaten eines Objekts keine sicherheitsrelevanten Informationen enthalten. Einzig das Objekt selbst muss vor unerlaubten Zugriffen geschützt werden.

Daher reicht es aus, das *value*-Feld direkt zu verschlüsseln, wodurch sich einerseits die Entschlüsselung bei erlaubten Zugriffen nur auf die betroffenen Datensätze beschränkt, andererseits können unerlaubte Zugriffe vorzeitig abgebrochen werden. Bei einem Aufruf von *getObject* ergibt sich beispielsweise folgendes Vorgehen:

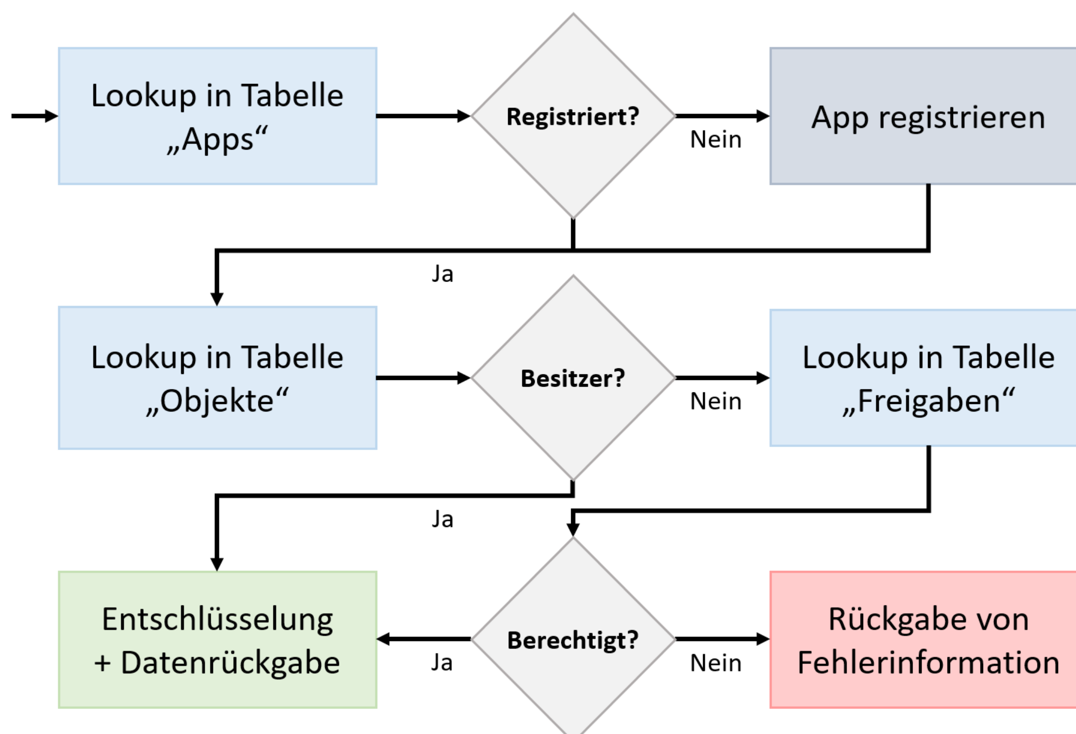


Abbildung 3.4: Zugriff auf den SOC am Beispiel einer Leseoperation

Dabei wird außerdem verhindert, dass ein Angreifer mit Schreibzugriff auf die Datenbank unbemerkt Änderungen an Objektdaten vornimmt, da hierzu der Schlüssel benötigt wird. Stattdessen könnte er jedoch das *owner*-Feld des Datensatzes manipulieren, um die Daten hinterher selbst über die Schnittstelle abzurufen. Um dies zu verhindern, müssen verschiedene Schlüssel für individuelle Apps vergeben werden, so dass Daten nicht durch „falsche Besitzer" entschlüsselt werden können (mehr dazu in Abschnit 4.4.4).

Eine weitere Möglichkeit, Zugriff auf ein Objekt zu erlangen, wäre das Anlegen einer entsprechenden Freigabe. Daher müssen diese ebenfalls verschlüsselt werden, damit garantiert wird, dass sie nur über den SOC erzeugt werden können. Besser wäre jedoch, ein zusätzliches Feld mit einer Prüfsumme über beide Werte verschlüsselt zu speichern, damit Anfragen „normal" durchgeführt werden können.

Da sich Lese- und Schreibanfragen an Objekte im SOC entweder auf einzelne oder eine definierte Liste von Datensätzen beschränken (siehe Abschnitt 3.1.1), müssen insgesamt deutlich weniger Daten entschlüsselt werden als bei Komplettverschlüsselung der Datenbank. Ebenfalls reicht es, den Vorgang nur in eine Richtung auszuführen (je nachdem, ob Daten gelesen oder geschrieben werden). Darüber hinaus können manche Operationen wie z.B. das Löschen von Objekten, Freigaben oder auch allen eigenen Daten (siehe Abschnitt 3.1.2) nun vollständig ohne Verschlüsselungs-Overhead durchgeführt werden.

3.4 Sicherheitsanalyse

Um zu bewerten, inwieweit das vorgestellte Konzept potentiellen Angriffsszenarien standhält, müssen diese zunächst definiert werden. Wichtig ist hierbei die Unterscheidung, ob ein Angreifer physikalischen Zugang zu einem Gerät besitzt (etwa durch Diebstahl) oder ob der Angriff nur über die auf dem Gerät befindliche Software ausgeführt wird. Die folgenden Beispiele basieren auf der Android-Plattform.

3.4.1 Physikalischer Angriff

Vidas et al. unterscheiden physikalische Angriffsszenarien wie folgt [VVC11]:

- **Physikalischer Zugriff mit aktiver ADB**
- **Physikalischer Zugriff ohne aktive ADB**
- **Physikalischer Zugriff ohne Gerätesperre**

ADB bezeichnet hierbei die *Android Debug Bridge* [dev17a], eine Schnittstelle, mit der über USB eine Verbindung zwischen dem Mobilgerät und einem Computer hergestellt werden kann. Im Gegensatz zum regulären Datenaustausch über USB besteht dabei die Möglichkeit, Software direkt auf dem Gerät auszuführen und Logdateien live abzugreifen. Wie der Name schon sagt,

handelt es sich primär um ein Entwickler-Tool zum Debuggen von Apps. Die Funktion muss einmalig in den Geräteeinstellungen aktiviert werden, außerdem wird ein kompatibler Treiber auf Computerseite benötigt.

Mit ADB-Verbindung ist es möglich, beliebige Apps auf einem Gerät zu installieren und auszuführen. Dies geschieht entweder direkt über die Kommandozeile oder die Benutzeroberfläche einer Android-Entwicklungsumgebung.

Dabei ist es auch möglich, Änderungen an Einstellungen vorzunehmen, wie etwa die Deaktivierung der Gerätesperre [Reh12]. Letzteres funktioniert jedoch nur auf älteren Android-Geräten, inzwischen verhindert das System den Zugriff auf die Debug Bridge bei gesperrtem Bildschirm, da zunächst ein Sicherheitsdialog bestätigt werden muss.

Alternativ dazu kann der Recovery-Modus des Geräts genutzt werden. Vidas et al. beschreiben, wie durch das Aufspielen eines eigenen Recovery-Images Zugriff auf die ADB-Kommandozeile erlangt werden kann [VZC11]. Der Recovery-Modus ist unabhängig vom normalen Bootvorgang des Geräts und wird daher nicht durch die Gerätesperre geschützt.

Insgesamt besteht demnach bei physikalischem Zugriff auf ein Gerät letztendlich die Möglichkeit, vollständigen Zugriff auf das System zu erlangen. Sobald die Gerätesperre umgangen wurde, kann das Gerät wie aus Anwendersicht bedient werden, wodurch sämtliche installierten Apps genutzt und dadurch deren im SOC gespeicherte Daten ausgelesen werden können.

Der einzige Schutz dagegen ist die grundlegende Verschlüsselung eines Geräts, wobei der Anwender vor der Nutzung ein Passwort eingeben muss [SFK+10]. Ohne Kenntnis des Passworts kann nicht mehr auf die gespeicherten Daten zugegriffen werden; lediglich ein *Factory Reset* ist möglich, wobei alle Daten verloren gehen. Ein solcher Schutz liegt jedoch nicht im Aufgabenbereich des SOC, sondern muss vom Anwender auf Geräteebene aktiviert werden.

3.4.2 Softwarebasierter Angriff

Im Folgenden liegt daher der Fokus auf softwarebasierten Angriffsszenarien, also „bösartigen“ Apps, die unerlaubt auf gespeicherte Daten zugreifen wollen. [CRP14] definiert hierbei drei Arten von Angreifern:

- **Malicious App Attacker** - Eine App ohne Root-Rechte, welche potentiell alle Android-Berechtigungen besitzt, die eine aus dem App-Store installierte App anfordern kann.
- **Root Attacker** - Ein Angreifer mit Root-Rechten, die entweder durch Ausnutzung einer Sicherheitslücke oder auf einem bereits gerooteten Gerät erlangt wurden. Apps mit Root-Zugriff können das vollständige Dateisystem des Geräts (inklusive Systembereich und App-internen Daten) auslesen und verändern.
- **Intercepting Root Attacker** - Ein solcher Angreifer hat Root-Zugriff und kann darüber hinaus Eingaben des Anwenders überwachen, beispielsweise auch Passwörter.

Anwender installieren solche Apps i.d.R. selbst, da sie im Vordergrund sinnvolle Funktionen bereitstellen, die die jeweiligen Berechtigungen rechtfertigen. Apps mit Root-Voraussetzung existieren sogar innerhalb der offiziellen App-Stores, ein bekanntes Beispiel hierfür ist der sogenannte *Root Explorer* oder auch *Titanium Backup* aus dem Google Play Store. Hierbei kann der Anwender nicht erkennen, ob im Hintergrund potentiell unerwünschte Aktionen auf Basis der gewährten Berechtigungen durchgeführt werden [XSA12].

Während Apps vor Freigabe im App-Store einer automatischen Prüfung unterzogen werden, garantiert dies nicht, dass jeder Missbrauch von Rechten dabei auch tatsächlich erkannt wird; im Gegenteil, im Google Play Store wurde beispielsweise schon mehrfach Malware entdeckt [HSG11]. Bei der Installation von Apps aus unbekannten Quellen *(Sideloading)* ist das Risiko jedoch exponentiell größer.

Betrachtet man die Sicherheitsmaßnahmen des SOC im Hinblick auf die zuvor genannten softwarebasierten Angriffsszenarien, gelangt man zu folgenden Ergebnissen:

Apps, die lediglich ein Übermaß an kritischen Berechtigungen besitzen, können darüber keinen Zugriff auf geschützte Daten des SOC erlangen, da der SOC nicht das traditionelle Android-Berechtigungsmanagement nutzt. Zugriff auf Daten kann generell nicht von Apps angefordert, sondern muss explizit von deren Besitzer erlaubt werden.

Prozesse mit Root-Zugriff können das interne Dateisystem des SOC auslesen und verändern. Aufgrund der in Abschnitt 3.3 beschriebenen Verschlüsselungsstrategie ist es ihnen jedoch nicht möglich, Änderungen an gespeicherten Datensätzen vorzunehmen oder neue Freigabeberechtigungen zu erzeugen. Lediglich das Löschen vorhandener Daten und Berechtigungen sowie der gesamten Datenbank ist möglich; dies ist jedoch ein grundlegendes Risiko bei allen Apps, da es technisch nicht verhindert werden kann, und kein konkreter Nachteil des SOC.

Können Eingaben des Anwenders überwacht werden, ist beispielsweise die Aufzeichnung des Entsperrungsmusters oder auch jeglicher Art von Passworteingabe möglich. Die Verschlüsselung des SOC darf daher nicht auf einer solchen Eingabe basieren (beispielsweise durch deterministische Ableitung des Schlüssels aus einem Passwort). Stattdessen muss garantiert werden, dass nur der SOC selbst in der Lage ist, auf den entsprechende Schlüssel zuzugreifen. Wie dies auf Android-Geräten möglich ist, wird in Abschnitt 4.4.3 beschrieben.

Insgesamt können Vertraulichkeit und Unversehrtheit der Daten im SOC demnach bei allen zuvor beschriebenden softwarebasierten Angriffsszenarien garantiert werden.

4 Implementierung

Das zuvor beschriebene Konzept des SOC soll auf Basis des Android-Betriebssystems implementiert werden. Android ist das seit Jahren am weitesten verbreitete mobile Betriebssystem und läuft aktuell auf 80,47% aller aktiven Smartphones [Sta17a].

Im Folgenden werden die Hauptkomponenten des SOC individuell betrachtet und Möglichkeiten für deren Implementierung auf Basis des Android-Betriebssystems diskutiert. Wichtig ist hierbei, dass sich der SOC in zwei grundlegende Teilbereiche spaltet: eine API und einen Service. Dies ist notwendig, da die Serialisierung aus technischen Gründen bereits „außerhalb" des Services stattfinden muss. Abbildung 4.1 zeigt eine stark vereinfachte Übersicht der in diesem Kapitel vorgestellten Komponenten:

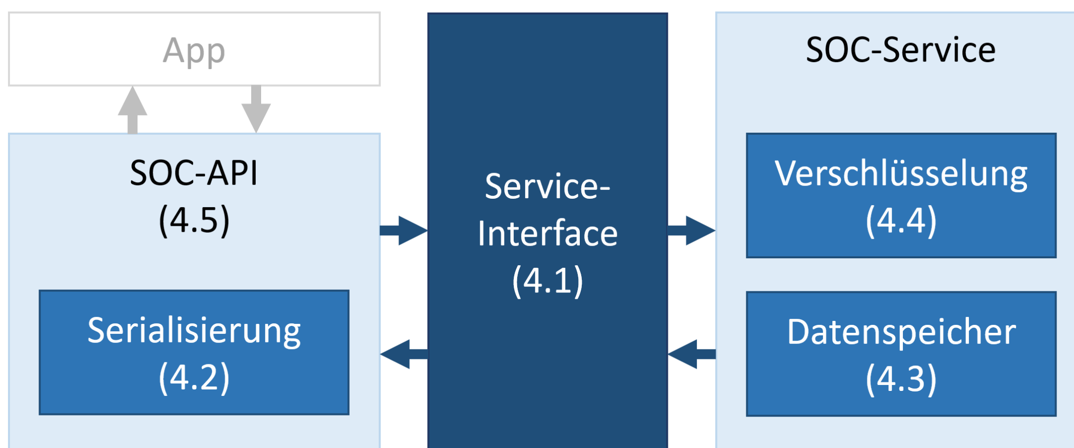


Abbildung 4.1: Übersicht der Hautpkomponenten des SOC

4.1 Service-Interface

Damit der SOC von allen installierten Apps angesprochen werden kann, muss er als *Service* implementiert werden. Dabei handelt es sich um eine App, die im Hintergrund des Betriebssystems läuft und keine eigene Benutzeroberfläche bereitstellt [dev17p]. Services können bei Bedarf von anderen Apps gestartet werden, die Verknüpfung zwischen App und Service ist über ein *Binding* möglich (siehe Abbildung 4.2).

Wie in Abschnitt 3.1.1 beschrieben, soll der Container eine Reihe von Methoden bereitstellen, über die alle Zugriffe abgehandelt werden. Um externen Prozessen Zugriff auf Methoden eines Services zu erlauben, definiert dieser ein Interface auf Basis der *Android Interface Definition Language (AIDL)* [dev17b].

AIDL erlaubt hierbei standardmäßig dieselben Parameter, die auch bei anderen prozessübergreifenden Nachrichten möglich sind; neben primitiven Datentypen sind *String* und *CharSequence* möglich, außerdem Kompositionen wie *List* und *Map*, sofern diese wiederum auf unterstützten Datentypen basieren. Parameter können als *in*, *out* oder *inout* deklariert werden.

Die Definition des Interfaces wird innerhalb des Services in Form einer .aidl-Datei abgelegt. Diese verzeichnet die Signaturen (Name und Parameter) aller aufrufbaren Methoden. Die Implementierung der jeweiligen Methoden wird im Service selbst definiert. Damit eine aufrufende App das Interface verwenden kann, muss sie eine exakte Kopie der .aidl-Datei besitzen.

Methodenaufrufe über das Service-Interface werden synchron durchgeführt. Um dadurch nicht die App zu blockieren, sollten sie aus einem separaten Thread (z.B. *AsyncTask*) heraus aufgerufen werden. Bei Datenabfragen wäre dann jedoch eine zusätzliche Strategie erforderlich, um diese an die aufrufende App zurückzugeben.

Zusätzlich zu den standardmäßig unterstützten Datentypen ist es mit AIDL zwar möglich, eigene Klassen zu importieren, hierfür muss die jeweilige Klasse jedoch *Parcelable* implementieren, also eine benutzerdefinierte Serialisierung definieren (siehe Abschnitt 4.2.1).

Darüber hinaus würde die Verwendung des SOC damit auf die dem Container bekannten Klassen eingeschränkt werden. Für eine von Objekttypen vollständig unabhängige Implementierung ist dieser Ansatz daher nicht anwendbar.

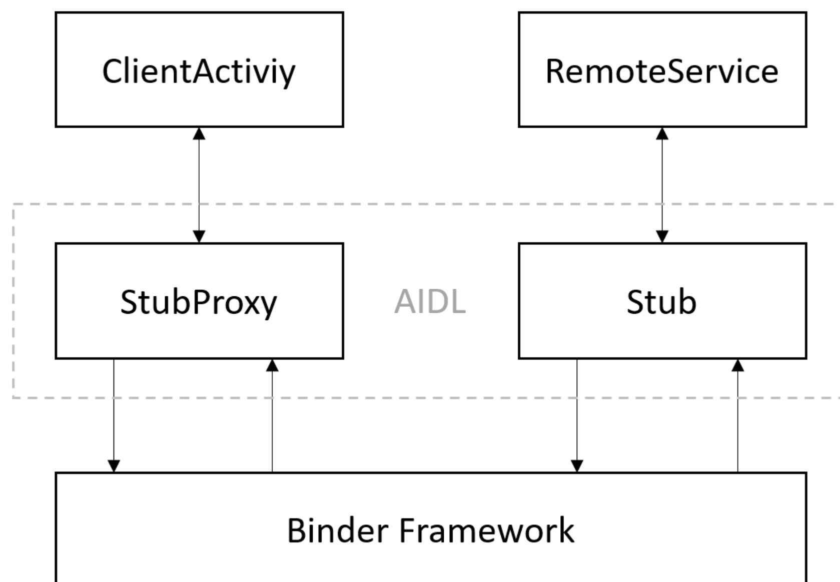


Abbildung 4.2: Kommunikation zwischen Client und Remote Service via AIDL-Interface

4.2 Serialisierung von Objekten

Wie im vorhergehenden Abschnitt beschrieben, sind für die prozessübergreifende Übertragung von Parametern nur bestimmte Datentypen zulässig. Für die Darstellung von Objekten bieten sich dabei Byte-Arrays oder Strings an, wobei es eine Vielzahl von Möglichkeiten gibt, eine entsprechende Abbildung zu definieren.

Im Folgenden werden zunächst die von Java und Android bereitgestellten nativen Serialisierungsverfahren beschrieben; danach werden zwei alternative Ansätze für die textbasierte Serialisierung nach XML und JSON vorgestellt.

4.2.1 Native Methoden

Serializable

Java bietet einen Standardmechanismus zur Umwandlung von Objekten in Byte-Streams. Voraussetzung ist dabei, dass das Objekt das Interface *java.io.Serializable* implementiert [Ora15]. Dabei wird *ObjectOutputStream* verwendet, um die Informationen in ein *OutputStream*-Objekt zu schreiben. Ein Beispiel hierfür ist *FileOutputStream*, falls die serialisierten Daten als Datei abgelegt werden sollen.

Enthält ein Objekt Referenzen auf andere komplexe Objekte, werden diese ebenfalls serialisiert und bei Deserialisierung wiederhergestellt. Falls ein verschachteltes Objekt *Serializable* nicht unterstützt, muss es als *transient* markiert werden, wodurch es beim Serialisierungsvorgang übersprungen wird.

Externalizable

Eine Alternative zur automatischen Java-Serialisierung bietet *java.io.Externalizable* [Ora17]. Hierbei muss der Entwickler der Objektklasse den Serialisierungsvorgang manuell definieren. Vorteil daran ist, dass nur tatsächlich benötigte Felder abgelegt werden können, während *Serializable* durch die Verwendung von *Java Reflection* einen Overhead an Metadaten mitbringt, etwa Strukturen von Superklassen und verknüpften Objekten [Gar14]. Darüber hinaus kann durch spezifische Deserialisierungs-Logik auf Änderungen der Klassenstruktur, beispielsweise in Verbindung mit „veralteten“ serialisierten Datensätzen, reagiert werden.

Eine entsprechende Klasse muss sowohl *Externalizable* als auch die Methoden *readExternal()* und *writeExternal()* implementieren, welche die betroffenen Felder und deren Reihenfolge festlegen. Ebenfalls muss ein parameterfreier Konstruktor existieren. *Externalizable* kann *Serializable* als Fallback verwenden, zum Beispiel für verschachtelte Objekte. Die Verwendung auf Basis von *ObjectOutputStream* ist analog zu *Serializable*.

Parcelable

Im Gegensatz zu den vorhergehenden Beispielen wurde *android.os.Parcelable* [dev17k] speziell für das Android-Betriebssystem entwickelt. Ein *Parcel* ist ein Container, mit dem Datensätze und Referenzen als Nachrichten zwischen Prozessen verschickt werden können. Ähnlich wie bei *SharedPreferences* (siehe Abschnitt 4.3.1) werden Methoden für eine Vielzahl von Datentypen bereitgestellt; ebenfalls ist es aber möglich, beliebige komplexe Objekte abzulegen, sofern diese *Parcelable* implementieren.

Wie bei *Externalizable* müssen entsprechende Klassen die Methoden zur Umwandlung manuell definieren. Hierbei wird *writeToParcel()* sowie ein Konstruktor auf Basis eines *Parcel*-Objekts benötigt. Dadurch kann die Größe des serialisierten Objekts auf die tatsächlich benötigten Daten eingeschränkt werden, was die Übertragungsgeschwindigkeit deutlich verbessert.

In der Android-Dokumentation wird jedoch explizit darauf hingewiesen, dass *Parcel* keinen sinnvollen Serialisierungsersatz für das persistente Speichern von Objekten darstellt, da nach Änderungen der Klassenimplementierung alte Datensätze möglicherweise nicht mehr korrekt eingelesen werden können [dev17j].

4.2.2 Implementierungsunabhängige Ansätze

Die zuvor beschriebenen Verfahren setzen eine Implementierung des jeweiligen Serialisierungsinterfaces als Teil der Klassenstruktur voraus. Sie sind somit nicht universell für komplexe Objekte einsetzbar. Der SOC muss jedoch – ähnlich wie der in Abschnitt 2.2.1 vorgestellte MetaService – in der Lage sein, beliebige Objekte zu verarbeiten.

Eine alternative Lösung sind daher Fremdbibliotheken, die Objekte unabhängig von ihrer Implementierung serialisieren können. Dies wird durch eine Darstellung in Textform bewerkstelligt, die lediglich die Felder des Objekts und deren Werte direkt abbildet, ohne zusätzlichen Struktur-Overhead. Ein Vorteil dieser Variante ist auch, dass die erzeugten Datensätze für Menschen lesbar sind.

XML

Ein bekanntes textbasiertes Datenformat ist *XML (Extensible Markup Language)*. Es unterstützt das Verschachteln von Werten und die Zuordnung von Metadaten durch Attribute. Ein bestehendes Framework für XML-Serialisierung von Java-Objekten ist *Simple XML Serialization*, beworben wird es mit „einfacher Bedienung“ und „hoher Performance“ [sim13].

Der Entwickler einer Klasse hat dabei die Möglichkeit, auf die serialisierte Form eines Objekts durch *Annotations* Einfluss zu nehmen; beispielsweise können bestimmte Felder als *@Element*, *@Attribute* oder *@Root* deklariert oder Tag-Namen manipuliert werden.

All dies ist jedoch optional; per Default werden die Objektklasse als Root-Element und die Felder als Kind-Elemente angelegt, Serialisierung und Deserialisierung funktionieren auch ohne Anpassung der Klasse problemlos. Listing 4.1 zeigt die Default-Abbildung für ein Objekt der Klasse *MedicalData* (Klassenstruktur siehe Abbildung 6.1).

Listing 4.1 Serialisierung eines Objekts der Klasse *MedicalData* mit *Simple XML*

```
<medicalData>
      <alt>-6.260827768246669E8</alt>
      <bsl>1361117995</bsl>
      <id>-2781882645812445755</id>
      <lat>7.006688423501501E8</lat>
      <lon>-4.2155819396284945E7</lon>
      <patient>
            <firstname>11419e5f-8055-411f-b679-aeed1269de77</firstname>
            <id>-2738918060703821946</id>
            <lastname>1e096c53-4e7b-4008-981b-f86763d5fdca</lastname>
            <patientdata>e0a3b653-04b6-4dee-9914-19ae7b9b5005</patientdata>
      </patient>
      <timestamp>9285c28e-44cf-4774-bc27-f94ee9ff1991</timestamp>
</medicalData>
```

JSON

Neben XML ist *JSON (JavaScript Object Notation)* ein weiteres textbasiertes Datenaustauschformat, welches in den letzten Jahren immer mehr an Beliebtheit gewonnen hat [tre17]. Gegenüber XML punktet es vor allem durch seine schlanke Darstellung, da keine End-Tags benötigt werden, was den Größe der Kontrollstrukturen effektiv halbiert.

Die wohl bekannteste Bibliothek für JSON-Serialisierung ist *GSON* [Goo17], die von Google entwickelt und seit 2008 als Open Source veröffentlicht wird. GSON stellt keine Voraussetzung an die Implementierung einer Klasse mit Ausnahme der Existenz eines parameterlosen Konstruktors.

Die Verwendung ist, verglichen mit den anderen hier vorgestellten Verfahren, sehr komfortabel: Serialisierung und Deserialisierung eines Objekts können jeweils mit einer einzigen Codezeile bewerkstelligt werden: durch Aufruf der Methoden *toJson(Object)* und *fromJson(String, Class)*. Listing 4.2 zeigt die Default-Abbildung für ein Objekt der Klasse *MedicalData*.

Neben der Default-Serialisierung kann auch bei GSON Einfluss auf das Ergebnis genommen werden, beispielsweise bezüglich Formatierung, Behandlung von *null*-Werten oder Ausschluss bzw. Einbeziehung bestimmter Feldarten (wie *static*, *transient*, *volatile*).

Im Gegensatz zum vorhergehenden Ansatz kann dies aber größtenteils über den Serialisierungsaufruf konfiguriert werden, ohne Zugriff auf die Klassenimplementierung zu erfordern. Darüber hinaus ist es sogar möglich, völlig eigenständige Serialisierer und Deserialisierer für bestimmte Klassen zu definieren.

Listing 4.2 Serialisierung eines Objekts der Klasse *MedicalData* mit *GSON*

```
{
      "alt": -626082776.8246669,
      "bsl": 1361117995,
      "id": -2781882645812445700,
      "lat": 700668842.3501501,
      "lon": -42155819.396284945,
      "patient": {
            "firstname": "11419e5f-8055-411f-b679-aeed1269de77",
            "id": -2738918060703822000,
            "lastname": "1e096c53-4e7b-4008-981b-f86763d5fdca",
            "patientdata": "e0a3b653-04b6-4dee-9914-19ae7b9b5005"
      },
      "timestamp": "9285c28e-44cf-4774-bc27-f94ee9ff1991"
}
```

4.2.3 Fazit

Der SOC muss jeden Objekttyp serialisieren können, unabhängig von dessen Implementierung. Daher wäre es ausreichend, eines der implementierungsunabhängigen Verfahren zu nutzen. Wenn eine Klasse jedoch bereits ein Interface für *Serializable* oder *Externalizable* anbietet, sollte dieses auch genutzt werden, da es i.d.R. die optimale Serialisierungsform für den jeweiligen Objekttyp erzeugt. *Parcelable* sollte hierbei jedoch ausgeschlossen werden, da keine Kompatibilität zwischen verschiedenen Versionen einer Klassenimplementierung garantiert werden kann (siehe Abschnitt 4.2.1).

Ein Ansatz auf Basis mehrerer Verfahren ist technisch dadurch lösbar, dass jedes Objekt zunächst auf Unterstützung eines nativen Serialisierungsinterfaces (*Serializable* oder *Externalizable*) überprüft wird. Nur wenn keines vorhanden ist, wird als Fallback einer der implementierungsunabhängigen Ansätze genutzt.

Das verwendete Verfahren muss dann zusammen mit dem Objekt gespeichert werden, hierzu wird das Schema der Objekt-Tabelle (siehe Abschnitt 3.2) um ein weiteres Feld „method“ erweitert. Beim Auslesen eines Objekts kann dies wiederum überprüft und das korrekte Verfahren zur Deserialisierung genutzt werden.

4.3 Datenspeicher

Der SOC muss Objekte persistent speichern, damit diese zu jeder Zeit allen berechtigten Apps zur Verfügung stehen. Da die Verwaltung der Daten ausschließlich durch den Container selbst stattfindet und Zugriffe über dessen bereitgestellte Methoden durchgeführt werden, besteht völlige Freiheit bei der Implementierung des darunterliegenden Datenspeichers. Android bietet hierbei verschiedene grundlegende Ansätze, die im Folgenden bewertet werden.

4.3.1 Shared Preferences

Android stellt die *SharedPreferences* API [dev17o] bereit, um Datensätze als Schlüssel/Wert-Paare innerhalb einer App zu speichern. Die App kann dabei verschiedene separate Dateien verwalten, die wiederum mehrere Datensätze enthalten dürfen. Die Dateien werden hierbei intern im XML-Format abgelegt. Schlüssel und Dateinamen müssen vom Typ *String* sein, Werte können dagegen die Datentypen *String*, *Boolean*, *Float*, *Integer* oder *Long* haben.

Der SOC könnte Objekte in serialisierter Form ablegen, wobei der Schlüssel dem Bezeichner des Datensatzes entspräche. Für jede im SOC verwaltete App würde dabei eine separate Datei mit deren Datensätzen erzeugt. Sonstige Metadaten, wie z.B. Freigabeinformationen, müssten jedoch in einer zusätzlichen Konfigurationsdatei abgelegt werden.

Da die Werte als Dateien abgelegt sind, existiert theoretisch kein Limit für die Größe der Datensätze – mit Ausnahme der Limits für den jeweiligen Datentyp und des maximal für die App verfügbaren Speicherplatzes. Da jedoch der Inhalt der Dateien nach dem ersten Öffnen im Hauptspeicher gecacht wird, kann das Speichern großer Datensätze zu einer Verschlechterung der Performance bis zum Speicher-Überlauf führen [Son14]. *SharedPreferences* wurden (entsprechend dem Namen) ursprünglich zur Ablage von Anwendereinstellungen konzipiert und sind daher eher für kleine Datenmengen vorgesehen.

4.3.2 Internes Dateisystem

Ohne den Umweg über *SharedPreferences* können Objekte in serialisierter Form auch direkt als Dateien gespeichert werden (vgl. Abschnitt 2.1.1). Da kein externer Direktzugriff auf die Daten möglich sein soll, kann der SOC das App-interne Dateisystem nutzen.

Im Gegensatz zum vorhergehenden Ansatz würde hierbei eine Datei pro Objekt erzeugt. Die Hierarchie eines Dateisystem kann dabei ausgenutzt werden, indem beispielsweise alle Datensätze einer App innerhalb eines entsprechenden Verzeichnisses abgelegt werden. Genau wie zuvor ist jedoch auch hier die manuelle Verwaltung einer Konfigurationsdatei nötig, die zusätzliche Metadaten zur Freigabeverwaltung etc. enthält.

4.3.3 Datenbank

Beide vorhergehenden Verfahren erfordern eine getrennte Verwaltung von Datensätzen und deren Metadaten. Während es zwar möglich wäre, die Metadaten direkt zu den Datensätzen hinzuzufügen (z.B. als CSV-Datenstruktur), wäre es dann erforderlich, Datensätze zunächst vollständig einzulesen, nur um deren Metadaten auszuwerten. Dies stellt einen unnötigen Overhead dar, weshalb die Trennung der Metadaten-Verwaltung dort sinnvoll ist.

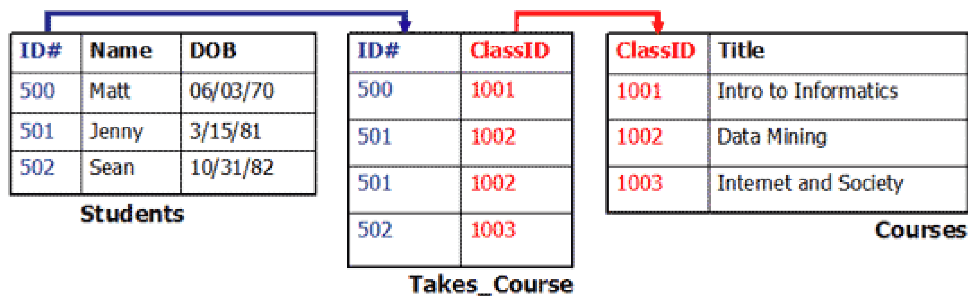


Abbildung 4.3: Beispiel einer relationalen Datenbank mit Schlüsselverweisen

Im Gegensatz zu „einfachen“ Dateien bzw. Schlüssel/Wert-Paaren bieten Datenbanken die Möglichkeit, Einträge mit mehreren Werten strukturiert abzuspeichern. Damit können Datensätze direkt durch zusätzlich benötigte Metadaten ergänzt werden. Datenbanksysteme stellen darüber hinaus Schnittstellen zur Verfügung, über die selektive Abfragen formuliert werden können. Dies reduziert den Entwicklungsaufwand für den SOC im Vergleich zur manuellen Verwaltung der Speicherstrukturen bei den vorhergehenden Ansätzen. Da für den SOC eine komplexere Datenverwaltung mit zusätzlichen Metadaten (siehe Abschnitt 3.2) benötigt wird, ist die Verwendung eines Datenbanksystems hier sinnvoll.

Relationale Datenbank

Relationale Strukturen waren viele Jahre lang der unangefochtene Standard für Datenbanksysteme [SMA+07]. Angelehnt an eine Tabelle stellt dabei jede „Zeile“ einen eigenen Datensatz dar, dessen einzelne Werte in „Spalten“ verteilt sind (siehe Abbildung 4.3). Das *Schema* (verfügbare Spalten) einer Tabelle ist dabei fest vorgegeben [Cod70].

Dies ermöglicht gezielte Abfragen auf Basis der Werte einzelner Spalten sowie das Anlegen von Indexlisten zur Performance-Steigerung (z.B. für bestimmte Sortierungen). Querverweise zwischen Tabellen sind durch Fremdschlüssel möglich. Relationale Datenbanksysteme stellen eine Schnittstelle bereit (meist auf Basis von SQL), mit der nahezu beliebig komplexe Abfragen auf Basis der abgelegten Tabellen und Datensätze formuliert werden können.

Hauptmerkmal relationaler Systeme ist die Einhaltung der *ACID*-Eigenschaften:

- **Atomicity** - Eine Transaktion wird entweder ganz oder gar nicht ausgeführt; bei Abbruch werden bereits durchgeführte Änderungen rückgängig gemacht
- **Consistency** - Jede Sequenz von Operationen hinterlässt einen konsistenten Datenzustand; insbesondere sind alle Integritätsbedingungen erfüllt
- **Isolation** - Verhinderung/Einschränkung der gegenseitigen Beeinflussung parallel ablaufender Transaktionen (verschiedene Abstufungen sind möglich)
- **Durability** - Nach erfolgreichem Abschluss einer Transaktion sind Änderungen dauerhaft in der Datenbank gespeichert (auch bei Ausfällen)

NoSQL-Systeme

Im Gegensatz zu relationalen Datenbanksystemen bildete sich in den letzten Jahren eine Art Gegenbewegung, die unter dem Namen *NoSQL (Not only SQL)* zusammengefasst wird [Pok13]. Der Begriff beschreibt keine konkrete Datenbankarchitektur, sondern eine Sammlung unterschiedlichster Ansätze. Ziel ist dabei kein „optimales" System, stattdessen ist jede Architektur für bestimmte Anwendungsszenarien optimiert und damit für andere Kontexte nur bedingt brauchbar. In Cloud-Umgebungen wird beispielsweise verbesserte Skalierbarkeit benötigt; dies wird durch Verteilung der Daten auf separate Server bzw. Knoten erreicht.

Dafür müssen jedoch Konsistenzanforderungen gelockert werden; NoSQL-Systeme verwenden als Gegenpool zu *ACID* den Begriff *BASE (Basically Available, Soft State, Eventually Consistent)* [RSS15], hierbei werden vorübergehend inkonsistente Zustände zugelassen, um Verfügbarkeit und Toleranz gegenüber Verbindungsausfällen zwischen den Knoten zu verbessern. Dieser Trade-off wurde erstmals in Brewers CAP-Theorem formuliert [Bre00].

Im Folgenden werden verschiedene grundlegende NoSQL-Architekturformen kurz vorgestellt und bezüglich ihrer Verwendbarkeit für den SOC bewertet [NPP13]:

- **Key/Value-Stores** enthalten neben dem Schlüssel nur ein einziges „Feld" für Daten, was maximale Skalierbarkeit durch horizontale Partitionierung erlaubt. Der SOC muss jedoch Abfragen auf Basis verschiedener Metadaten durchführen können.
- **Wide-Column Stores** erlauben mehrere Attribute pro Schlüssel, wobei diese nicht global festgelegt sind. Optimiert sind die Systeme jedoch für spaltenbasierte Abfragen, wie z.B. Aggregationsoperationen.
- **Document Stores** speichern Datensätze in Form von strukturiertem Text (z.B. JSON, XML), wobei jedes Dokument ein eigenes Schema definiert. Datensätze verbrauchen daher deutlich mehr Speicherplatz als in einer relationalen Darstellung.
- **Graph Databases** speichern Informationen in Form eines Netzwerks. Der Fokus liegt hierbei auf der Berechnung knotenübergreifender Verbindungen, beim SOC werden Datensätze aber maximal einmal weiterverteilt.
- **Object Stores** können Objektstrukturen direkt ablegen und auslesen. Für den SOC ist dies jedoch ungeeignet, da Datensätze nicht in Objektform zwischen Prozessen übertragen werden können (siehe Abschnitt 4.1).

Insgesamt stellt im Hinblick auf den SOC keiner der betrachteten Ansätze eine Verbesserung gegenüber der Verwendung einer relationalen Datenbank dar. Vorteile der Skalierbarkeit können beim lokalen Speichern auf Mobilgeräten ebenfalls nicht ausgenutzt werden.

Betrachtet man die in Abschnitt 4.2.2 vorgestellten Verfahren, könnte die Implementierung auf Basis eines entsprechenden Document Stores in Betracht gezogen werden; dies verhindert jedoch den Einsatz einer variablen Serialisierungsstrategie (siehe Abschnitt 4.2.3).

4.4 Verschlüsselung

Um die im SOC abgelegten Datensätze vor unberechtigten Zugriffen zu schützen, ist eine Verschlüsselung nötig. Wie in Abschnitt 3.3 beschrieben, muss hierbei jedoch nicht die gesamte Datenbank verschlüsselt werden, da dies zu einem unnötigen Overhead bei Zugriffen führt; vielmehr reicht es, die tatsächlichen Datensätze zu schützen. Im Folgenden werden Ansätze diskutiert, wie eine solche Verschlüsselungsstrategie implementiert werden kann.

4.4.1 Verfahren

Grundsätzlich existieren zwei Arten von Verschlüsselung: *symmetrisch* und *asymmetrisch* [Sim79]. Bei symmetrischer Verschlüsselung wird eine Information durch Anwendung eines Schlüssels unkenntlich gemacht; derselbe Schlüssel kann später verwendet werden, um die Daten wiederherzustellen. Entsprechend müssen alle Instanzen, die Zugriff auf die Information erlangen sollen, den Schlüssel mitgeteilt bekommen. Dies ist gleichzeitig der große Nachteil dieses Ansatzes: Gerät der Schlüssel in falsche Hände, können verschlüsselte Daten von Dritten ausgelesen werden.

Asymmetrische Verschlüsselung löst dieses Problem, indem zwei verschiedene Schlüssel erzeugt werden: ein öffentlicher (*Public Key*) und ein geheimer (*Private Key*). Der *Public Key* darf hierbei frei verteilt werden; dieser kann allerdings nur genutzt werden, um Daten zu verschlüsseln. Um die Daten dagegen wiederherzustellen, ist der *Private Key* nötig. Dies stellt sicher, dass nur der beabsichtigte Empfänger die Daten lesen kann. Asymmetrische Verfahren sind daher deutlich sicherer, benötigen aber auch einen höheren Rechenaufwand.

Beim SOC besteht die Problematik der sicheren Schlüsselweitergabe nicht, da der Container selbst für Verschlüsselung und Entschlüsselung der Daten zuständig ist. Demnach ist ein symmetrischer Schlüssel hier ausreichend; es muss allerdings sichergestellt werden, dass dieser nicht von Dritten ausgelesen werden kann (siehe Abschnitt 4.4.3).

Android beinhaltet das Paket *javax.crypto* [dev17i], welches eine Reihe von Verschlüsselungsalgorithmen unterstützt, unter anderem die symmetrischen Verfahren *AES (Advanced Encryption Standard)* und *Blowfish*. *AES* (ursprünglich *Rijndael*) wurde 2001 als Nachfolger des als nicht mehr sicher erachteten *DES (Data Encryption Standard)* festgelegt. Unterstützt werden Schlüssel der Länge 128, 192 oder 256 Bit.

Blowfish existiert bereits seit 1993 und wurde ebenfalls als Alternative zu DES entwickelt; hierbei sind Schlüssel der Länge 32–448 Bit möglich. Während AES einen generell höheren Bekanntheitsgrad genießt, zeigte Blowfish in Tests eine bessere Performance [Tam08]. Beide Verfahren werden nach heutigem Stand als sicher eingestuft, wobei die Wahl des Schlüssels ausschlaggebend ist (siehe Abschnitt 4.4.2).

4.4.2 Wahl des Schlüssels

Verschlüsselungsverfahren, die noch nicht „geknackt“ werden konnten, sind sicher, solange der verwendete Schlüssel eine ausreichende Länge hat und keinerlei Information über die Zusammensetzung des Schlüssels bekannt ist. Grund dafür ist, dass solche Verfahren ausschließlich durch Brute-Force-Angriffe umgangen werden können. Ein solcher Angriff testet sämtliche Kombinationen von möglichen Zeichen als Passwort, bis das richtige gefunden wird.

Würde eine Verschlüsselung beispielsweise nur ein 4 Bit langes Passwort verwenden, gäbe es lediglich $2^4 = 16$ mögliche Kombinationen; demnach könnten die Daten trotz des eigentlich sicheren Verfahrens schnell ausgelesen werden. Die Anzahl der Kombinationen verdoppelt sich dabei mit jedem zusätzlichen Bit.

Aufgrund der stetig wachsenden Rechenleistung von Computersystemen muss daher eine Schlüssellänge definiert werden, bei der die Anzahl der Kombinationen weit über allem liegt, was aktuelle und zukünftige Rechner in einem „sinnvollen“ Zeitfenster berechnen können.

Arora rechnet vor, dass der schnellste Supercomputer der Welt (Stand 2012) etwa 10^{18} (eine Quintillion) Jahre benötigen würde, um einen Schlüssel mit 128 Bit zu knacken [Aro12]. Daher werden noch längere Schlüssel (z.B. 256 Bit) kaum als Sicherheitszuwachs angesehen.

Sobald ein Angreifer jedoch die Möglichkeit hat, Rückschlüsse auf die Zusammensetzung (auch in Teilen) des Schlüssels zu ziehen, etwa wenn dieser aus einem alphanumerischen Passwort berechnet wurde, ist die Sicherheit des Verfahrens stark eingeschränkt, da deutlich weniger Kombinationen ausgewertet werden müssen. Daher ist es wichtig, dass der Schlüssel möglichst zufällig generiert wird.

4.4.3 Schlüssel sicher ablegen

Sofern der Schlüssel eine ausreichende Länge und Komplexität besitzt, kann ein Angreifer nur Zugriff auf die Daten erlangen, indem der Schlüssel direkt ausgelesen wird. Der SOC muss den Schlüssel jedoch persistent speichern, um bei Datenzugriffen kryptografische Operationen ausführen zu können. Dies stellt ein Henne-Ei-Problem dar, da der abgelegte Schlüssel wiederum vor unberechtigten Zugriffen auf das App-interne Dateisystem geschützt werden muss.

Android stellt hierfür seit API-Level 18 einen *KeyStore* [dev17c] bereit, der von Apps genutzt werden kann, um eigene Schlüssel zu speichern. Unterschieden werden hierbei zwei grundlegende Anwendungsfälle:

Die *KeyChain* API ermöglicht die Verwaltung systemweit verfügbarer Zugangsdaten, wobei der Anwender in jedem Fall entscheiden muss, ob angeforderte Berechtigungen gewährt werden dürfen. Dem gegenüber steht der *Keystore Provider*, bei dem nur App-eigene Keys abgerufen werden können. Für die Verwendung im SOC ist nur die zweite Variante sinnvoll.

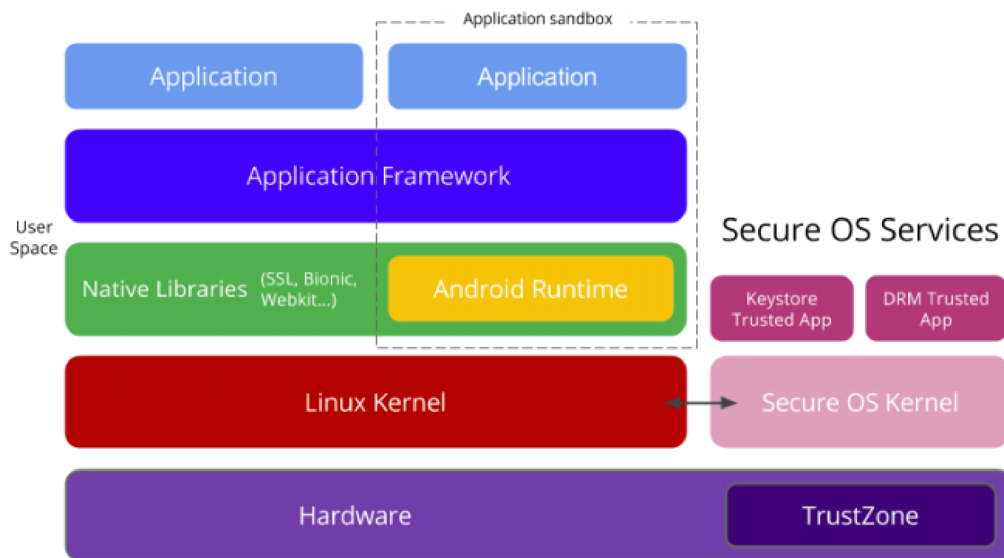


Abbildung 4.4: Sicherheitsarchitektur mit TrustZone-Unterstützung [Tah16]

KeyStore verhindert das unberechtigte Auslesen von Schlüsseln mit einer Reihe von Maßnahmen: Bei Verwendung eines Schlüssels zur Durchführung von kryptografischen Operationen wird dieser niemals an die aufrufende App weitergegeben, vielmehr wird die Operation innerhalb eines eigenen dafür vorgesehenen Systemprozesses durchgeführt.

Auf Geräten mit passender Hardware wird darüber hinaus ein spezieller „sicherer“ Speicherbereich verwendet, auf den lediglich bestimmte, als vertrauenswürdig deklarierte Apps Zugriff haben, um den tatsächlichen Schlüssel abzulegen (z.B. *TrustZone*, siehe Abbildung 4.4). Dieser ist physikalisch vom restlichen Speicherbereich getrennt und kann daher weder von unberechtigten Apps noch von Root-Zugriffen beeinträchtigt werden [CRP14].

Auf Geräten ohne entsprechende Hardware wird diese Funktionalität lediglich auf Softwareebene emuliert, daher kann die vollständige Sicherheit der Daten in diesem Fall nicht garantiert werden. Eine entsprechende Überprüfung ist mit *KeyInfo.isInsideSecurityHardware()* möglich.

Die Zugriffserlaubnis auf Schlüssel wird außerdem mit der Authentifizierung des Anwenders verknüpft, etwa durch Entsperrungsmuster oder PIN. Selbst auf Geräten ohne Hardwareunterstützung kann ein Schlüssel daher nur von einem Angreifer ausgelesen werden, wenn dieser Root-Zugriff besitzt, das Gerät entweder bereits entsperrt ist oder kein Entsperrungsschutz eingerichtet ist oder der Angreifer in der Lage war, eine entsprechende Eingabe aufzuzeichnen.

Ebenfalls werden Operationen zur Generierung von Schlüsseln für bestimmte Einsatzzwecke angeboten, was den Vorteil hat, das sich der Schlüssel niemals außerhalb der sicheren Umgebung (etwa im Arbeitsspeicher des Geräts) befindet. Darüber hinaus kann der Inhaber eines Schlüssels individuell einschränken, für welche Operationen der Schlüssel überhaupt genutzt werden darf. Insgesamt ist der Android *KeyStore* daher die bestmögliche Variante, Schlüssel einer App sicher zu verwalten.

4.4.4 Sicheres Löschen

Obwohl die Daten des SOC nur verschlüsselt gespeichert werden, muss aus Sicherheitsgründen die Möglichkeit bestehen, diese endgültig unleserlich zu machen bzw. zu löschen. Das kann sich sowohl auf einzelne Datensätze als auch auf die gesamte Datenbank beziehen. Wichtig ist dies, da die üblichen *DELETE*-Operationen von Datei- und Datenbanksystemen lediglich Verweise auf Datensätze löschen bzw. deren allokierten Speicher freigeben. Dies ist jedoch keine Garantie dafür, dass sie nicht mehr ausgelesen werden können, im Gegenteil: Solange der entsprechende physikalische Speicherbereich nicht von einem anderen Datensatz überschrieben wurde, ist es möglich, die Daten wiederherzustellen.

Ein möglicher Ansatz ist hierbei das Löschen des Schlüssels selbst: Selbst wenn die Datensätze nach wie vor im physikalischen Speicher liegen, können sie nicht mehr entschlüsselt werden, was sie wertlos macht, sofern der Schlüssel ausreichende Komplexität besitzt (siehe Abschnitt 4.4.2). Der Nachteil ist jedoch, dass dies alle Datensätze gleichermaßen unbrauchbar macht.

Alternativ zu einem einzigen (globalen) Schlüssel wäre es daher denkbar, für jede App einen eigenen Schlüssel zu generieren, mit dem deren Daten verschlüsselt werden. Dies würde das sichere Löschen aller Datensätze einer einzelnen App ermöglichen, ohne dass der restliche Inhalt des Containers beeinträchtigt wird (siehe Abschnitt 3.1.2).

Möglich wäre hierbei auch, dies automatisch mit der Deinstallation einer App zu verknüpfen. Der SOC müsste hierfür einen *BroadcastReceiver* registrieren, der auf Intents der Art *ACTION_PACKAGE_REMOVED* reagiert. Dabei wird der Name der deinstallierten App mit übergeben, was die Löschung des dazugehörigen Schlüssels auslösen könnte.

Um individuelle Objekte zu löschen, wäre dieser Ansatz dagegen zu umständlich. Hierbei kann stattdessen sichergestellt werden, dass Speicherbereiche von Datensätzen bei Löschoperationen nicht nur deallokiert, sondern manuell überschrieben werden. Möglich ist dies beispielsweise mit SQLite, durch entsprechende Konfiguration der Datenbank über *PRAGMA*-Anweisungen. Ist *PRAGMA schema.secure_delete* aktiviert, werden durch Löschoperationen entfernte Daten mit Nullen überschrieben [sql17].

Alle zuvor beschriebenen Ansätze funktionieren jedoch nur, wenn der Anwender oder eine verknüpfte App selbst die Anweisung zur Löschung gibt. Wird beispielsweise das Gerät gestohlen, hat der Anwender keinerlei Möglichkeit mehr, die Daten unleserlich zu machen.

Geambasu et. al. beschreiben eine Lösung dieses Problems; die Idee ist hierbei, den Schlüssel nicht auf dem Gerät, sondern ausschließlich auf einem externen Server zu speichern [GJG+11]. Der Vorteil daran ist, dass der Anwender bei Verlust des Geräts noch immer die Möglichkeit hat, den Schlüssel zu löschen. Dem gegenüber steht jedoch die Unsicherheit eines externen Faktors, etwa bezüglich Verfügbarkeit, Vertrauenswürdigkeit und Übertragungssicherheit.

4.5 SOC-API

Um App-Entwicklern die Anbindung an den SOC-Service möglichst einfach zu machen, wird eine Bibliothek mit allen Methoden bereitgestellt, die für den Zugriff benötigt werden. Da außerdem die Übergabe eines Objekts an einen Service nur in serialisierter Form stattfinden kann (siehe Abschnitt 4.1), muss dies bereits „außerhalb" des Services durchgeführt werden. Abbildung 4.5 zeigt schematisch den Zugriff auf den SOC über eine bereitgestellte API.

Natürlich kann eine App den SOC auch direkt über dessen Service-Interface ansprechen, jedoch erhöhen sich dadurch Entwicklungsaufwand und Fehlerrisiko deutlich. Eine Bibliothek kann dagegen zentral gepflegt und als Abhängigkeit in beliebige Android-Apps eingebunden werden (etwa via Maven oder als .jar-Datei). Angelehnt ist dieser Ansatz an den des in Abschnitt 2.2.1 beschriebenen MetaService.

4.5.1 Basisfunktionalität

Um alle grundlegenden Funktionen des SOC bereitzustellen, müssen dessen Interface-Methoden auf die API abgebildet werden (siehe Abbildung 4.5: *API-Wrapper*). Diese sollen von der aufrufenden App direkt verwendbar sein, ohne dass auf die interne Struktur des SOC Rücksicht genommen werden muss.

Der Methode *putObject()* wird dabei beispielsweise lediglich ein komplexes Objekt sowie dessen Schlüssel übergeben. Um das Objekt dann im SOC zu speichern, wird es von der API zunächst serialisiert und dann über die tatsächliche Interface-Methode des SOC-Service weitergegeben. Umgekehrt können empfangene Objekte nach der Deserialisierung direkt in Objektform an die App übergeben werden.

Serialisierung

Methoden zur Serialisierung und Deserialisierung können API-intern implementiert werden, da sie nicht direkt von den Apps aufgerufen werden müssen. Bei Übergabe eines Objekts wird auf Basis von dessen Klassenstruktur die entsprechende unterstützte Methode gewählt (siehe Abschnitt 4.2.3), wobei GSON als Fallback für Objekte ohne Serialisierungsinterface genutzt werden kann.

Die gewählte Methode sowie der vollständige Klassenname des Objekts müssen zusammen mit den serialisierten Objektdaten im SOC gespeichert werden. Umgekehrt werden bei der Deserialisierung Methode und Objektklasse benötigt, um eine neue Instanz des gespeicherten Objekts zu erzeugen. Wird der SOC dagegen auf Basis einer einzigen Serialisierungsmethode implementiert, ist das Speichern der Methode nicht notwendig.

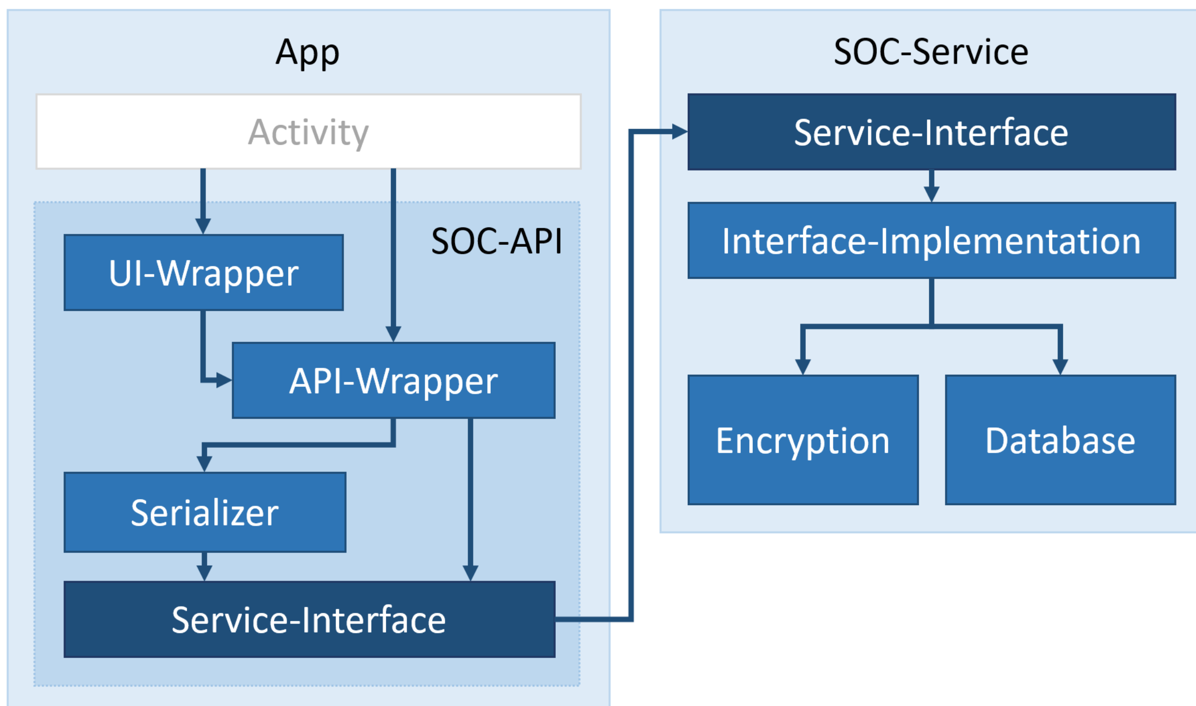


Abbildung 4.5: Kernkomponenten und Zugriffspunkte von SOC-API und SOC-Service

Service-Schnittstelle

Die Implementierung des Service-Interfaces ist ebenfalls Teil der Bibliothek, so dass die aufrufende App davon keine Kenntnis haben muss. Dazu muss die AIDL-Definition nur innerhalb der Bibliothek abgelegt werden, was eine zentrale Anpassung bei Änderung des Interfaces ermöglicht.

Um den Service anzusprechen, muss ein *AsyncTask* verwendet werden, der die Stub-Methoden des Interfaces mit den benötigten Parametern aufruft. Da ein solcher Aufruf keinen (synchronen) Rückgabewert für die aufrufende App produzieren kann, muss zur Datenrückgabe ein weiteres Interface definiert werden, dessen Methoden von der jeweiligen App implementiert werden können (mehr dazu in Abschnitt 5.2.2).

4.5.2 User Interface

Mit den zuvor beschriebenen Methoden kann eine App alle Funktionen des SOC über direkte Funktionsaufrufe nutzen und muss keinerlei eigene Logik für die Serialisierung und das Service-Interface implementieren. Um als Anwender mit dem SOC zu interagieren, ist jedoch eine grafische Benutzeroberfläche sinnvoll; diese kann ebenfalls als Teil der Bibliothek bereitgestellt werden, um Entwicklungsaufwand für die Apps zu reduzieren.

Dialog-Wrapper

Die Basismethoden des API-Wrappers können zur einfacheren Eingabe in Dialoge gekapselt werden (siehe Abbildung 4.5: *UI-Wrapper*). Dies erlaubt dem Anwender beispielsweise, beim Speichern einen Dateinamen einzugeben und beim Abrufen oder Teilen eines Datensatzes aus einer Liste verfügbarer Objekte und Apps auszuwählen. Löschoperationen kann dagegen eine Sicherheitsabfrage vorgeschaltet werden.

Für derartige Auswahldialoge muss das Service-Interface um zusätzliche Methoden erweitert werden, die beispielsweise eine Liste der eigenen oder durch Teilen verfügbaren Objekte abrufen. Der Dialog muss dafür bereits vor der Anzeige eine Anfrage an den SOC-Service stellen, was nur sinnvoll ist, wenn die entsprechende Latenz vernachlässigbar ist.

Nach Eingabe/Auswahl über einen Dialog wird die zugrundeliegende API-Wrapper-Methode mit den jeweiligen Parametern aufgerufen. Vorteil daran ist, dass das Fehlerpotential reduziert wird, da beispielsweise keine nicht-existenten Datensätze angesprochen werden und Löschungen nicht unbeabsichtigt stattfinden.

Fehlerbehandlung

Sowohl innerhalb der Bibliothek als auch beim Aufruf des SOC-Service können aus verschiedensten Gründen Fehler auftreten. Damit der Anwender darauf reagieren kann, müssen diese in einer für Menschen lesbaren Form dargestellt werden. Die Bibliothek definiert daher eine Liste von typischen Fehlerfällen, für die eine Meldung hinterlegt wird.

Die Interface-Methoden des SOC geben dementsprechend einen Zahlencode zurück, wenn ein Fehler auftritt, der dann von der Bibliothek auf eine Fehlermeldung abgebildet wird. Wird die aktuelle *Context*-Instanz der aufrufenden App an die API übergeben, kann die entsprechende Fehlermeldung direkt als Dialog angezeigt werden.

4.5.3 Fazit

Durch die Kombination aus SOC-Service und SOC-API ist es möglich, einen für App-Entwickler einfach zu verwendenden Datenspeicher für beliebige Objekte bereitzustellen. Hierbei werden Implementierungsdetails wie beispielsweise Serialisierungsmethode oder Service-Interface vollständig vor der aufrufenden App verborgen, so dass interne Änderungen am SOC möglich sind, ohne dass verknüpfte Apps ihre Schnittstellen anpassen müssen.

Die Implementierung einer Benutzeroberfläche verbessert darüber hinaus die Benutzerfreundlichkeit hinsichtlich der Bedienung durch den Anwender und verringert das Risiko fehlerhafter Aufrufe. Die Verwendung ist jedoch optional, bei Bedarf kann eine App stattdessen auch eigene Bedienelemente bereitstellen.

5 Prototyp

In diesem Kapitel wird der Prototyp des SOC vorgestellt, der als *Proof-of-concept* basierend auf den in Kapitel 4 betrachteten Ansätzen realisiert wurde. Anstatt ihn jedoch als eigenständigen App-Service zu implementieren, wurde er in die *Privacy Management Platform* integriert, da deren Aufbau die für den SOC benötigten Schnittstellen zwischen Apps, API-Bibliothek und Service-Komponente bereits zur Verfügung stellt.

Die folgenden Abschnitte beschreiben zunächst die für die Implementierung des SOC relevanten Elemente der Privacy Management Platform sowie die Komponenten, die der SOC dafür zusätzlich bereitstellen muss. Danach wird dargestellt, inwieweit der SOC-Prototyp die im vorherigen Kapitel beschriebenen Ansätze umsetzt.

5.1 Die Privacy Management Platform

Die *Privacy Management Platform (PMP)* [SM13; SM14] ist ein alternatives Berechtigungssystem für Android. Im Gegensatz zum traditionellen Berechtigungsmanagement werden hierbei Berechtigungen nicht den Apps, sondern lediglich deren individuellen Komponenten bzw. Features zugewiesen. Der Widerruf einzelner Berechtigungen deaktiviert damit lediglich das entsprechende Feature, während der Rest der App normal genutzt werden kann.

Die PMP bietet darüber hinaus die Möglichkeit, Berechtigungen in verschiedenen Abstufungen und abhängig von Kontextinformationen zu gewähren; dies ist für die Implementierung des SOC jedoch nicht von Relevanz und wird daher nicht genauer betrachtet.

5.1.1 Komponenten

Die PMP nutzt keine Elemente des traditionellen Android-Berechtigungssystems, sondern stellt eigene Verwaltungskomponenten bereit. Sie kann daher nur mit Apps genutzt werden, die die entsprechenden Schnittstellen definieren. Abbildung 5.1 zeigt das Zusammenspiel der einzelnen Komponenten des Berechtigungsmodells.

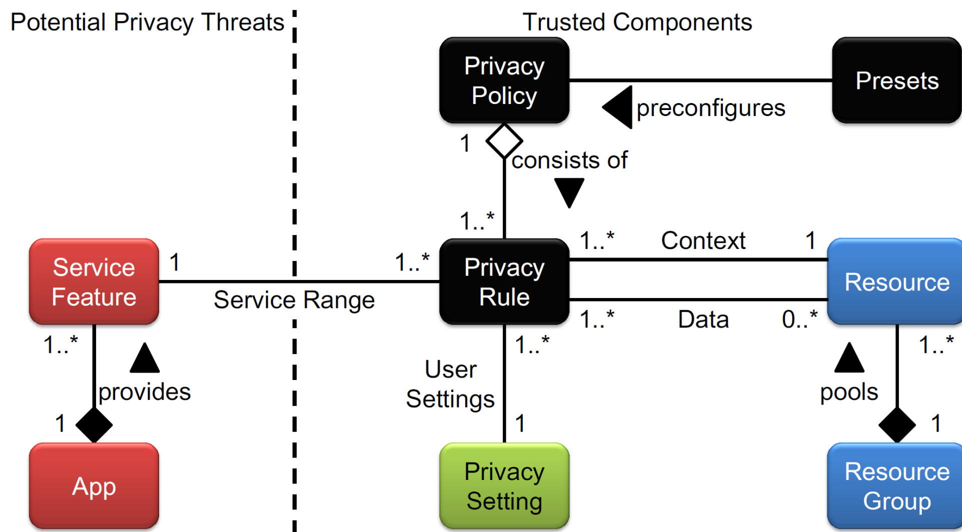


Abbildung 5.1: Berechtigungsmodell der Privacy Management Platform [SM14]

Ressourcen

Ressourcen dienen als Vermittler zwischen PMP-Apps und geschützten Daten. Eine Ressource kann dabei Zugriffe auf verschiedenste Softwareelemente (z.B. Kontakte, Kalender) oder Hardware-Sensoren (z.B. GPS) verwalten, wobei jede Ressource nur für eine einzige Datenquelle verantwortlich ist. Verwandte Ressourcen können in sogenannten *Ressourcengruppen* zusammengefasst werden.

Die Entwicklung und Installation von Ressourcen geschieht hierbei unabhängig von der PMP selbst: Alle verfügbaren Ressourcen sind in einem zentralen Archiv abgelegt und können bei Bedarf individuell über die PMP nachinstalliert werden. Das Bereitstellen von Updates für bereits existierende Ressourcen wird dadurch ebenfalls vereinfacht.

Eine Ressource kann verschiedene Modi zur Rückgabe von Daten anbieten: *korrekt*, *verändert*, *zufällig* und *leer*. Woher die ursprünglichen Daten (z.B. Standort) kommen bzw. wie diese jeweils erzeugt werden, ist dem Entwickler der jeweiligen Ressource überlassen.

Service Features und Privacy Settings

Jede PMP-kompatible App unterteilt die verfügbaren Funktionen in sogenannte *Service Features*. Jedes Feature muss dabei unabhängig von den anderen funktionsfähig sein und definieren, welche Ressourcen mit welchen Berechtigungen benötigt werden.

Der Anwender kann jeweils entscheiden, ob er die angeforderten Zugriffsberechtigungen gewähren möchte. Diese Festlegung wird in Form von *Privacy Settings* gespeichert. Neben dem einfachen Aktivieren bzw. Deaktivieren einer Berechtigung können dabei auch Abstufungen festgelegt werden, sofern die Ressource diese anbietet.

5.1.2 Technischer Aufbau

Um Apps mit der PMP kompatibel zu machen oder Ressourcen dafür zu entwickeln, müssen bestimmte Schnittstellen implementiert bzw. Strukturvorgaben befolgt werden. Die folgenden Angaben basieren auf der Implementierung des PMP-Protoyps, auf dessen Basis der SOC-Prototyp entwickelt wurde.

Apps

Damit eine installierte App über die PMP verwaltet werden kann, muss sie bei dieser registriert sein. Hierzu muss eine Referenz auf die PMP *RegistrationActivity* im Manifest der App als *LAUNCHER* deklariert werden (siehe Listing 5.1). Dadurch wird garantiert, dass die Registrierung beim ersten Start der App stattfindet.

Listing 5.1 Registrierungsaufruf einer PMP-App (AndroidManifest.xml)

```
<activity
      android:name="de.unistuttgart.ipvs.pmp.api.gui.registration.RegistrationActivity"
      ...
      <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
      ...
</activity>
```

Um darüber hinaus auf von der PMP bereitgestellte Ressourcen zuzugreifen, ist eine Verbindung mit dem PMP *AppService* nötig (siehe Listing 5.2). Hierbei muss außerdem das Package der Komponente angegeben werden, die mit dem Service kommunizieren soll.

Listing 5.2 Serviceaufruf einer PMP-App (AndroidManifest.xml

```
<service android:name="de.unistuttgart.ipvs.pmp.service.app.AppService" >
      <intent-filter>
            <action android:name="de.unistuttgart.ipvs.pmp.apps.testbench.write.soc" />
      </intent-filter>
</service>
```

Die Deklaration der Service Features wird dagegen in einer separaten Datei unter *assets/ais.xml* abgelegt. Diese definiert das *appInformationSet* (siehe Listing 5.3), in dem Name und Beschreibung von App und Service Features in der Form angegeben werden, in der sie nachher über die PMP-Verwaltungsoberfläche aufgelistet werden. Jedes Service Feature definiert außerdem die Liste der dafür benötigten Ressourcengruppen und Privacy Settings.

Listing 5.3 Informationsset einer PMP-App (ais.xml)

```
<appInformationSet>
      <appInformation>
            <name lang="en">SOC_Testbench</name>
            ...
      </appInformation>
      <serviceFeatures>
            <serviceFeature identifier="benchmark">
                  <name lang="en">SOC Testbench</name>
                  ...
                  <requiredResourceGroup
                        identifier="de.unistuttgart.ipvs.pmp.resourcegroups.soc"
                        minRevision="1">
                        <requiredPrivacySetting
                            identifier="useSOC">true</requiredPrivacySetting>
                  </requiredResourceGroup>
            </serviceFeature>
      </serviceFeatures>
</appInformationSet>
```

Ressourcen

Im Prototyp der PMP werden Ressourcengruppen als Apps implementiert, die wiederum mehrere einzelne Ressourcen in Form von Packages bereitstellen. Analog zum *appInformationSet* muss jede Ressourcengruppe ein *resourceGroupInformationSet* (siehe Listing 5.4) bereitstellen, das die verfügbaren PrivacySettings definiert.

Listing 5.4 Informationsset einer PMP-Ressourcengruppe (rgis.xml)

```
<resourceGroupInformationSet>
      <resourceGroupInformation
            identifier="de.unistuttgart.ipvs.pmp.resourcegroups.soc"
            icon="res/drawable/icon.png"
            className="SOCResourceGroup">
            <name lang="en">SOC Resource Group</name>
            ...
      </resourceGroupInformation>
      <privacySettings>
            <privacySetting
                  identifier="useSOC"
                  validValueDescription="'true', 'false'"
                  requestable="true">
                        <name lang="en">Use the SOC</name>
                  ...
            </privacySetting>
            ...
      </privacySettings>
</resourceGroupInformationSet>
```

Die Zuordnung, welches PrivacySetting von welcher Ressource benötigt wird, geschieht dagegen im Quellcode der Ressource selbst. Hierbei müssen die verfügbaren Ressourcen und PrivacySettings zunächst bei der PMP registriert werden. Entsprechende Methoden werden von einer Bibliothek namens *PMP-API* bereitgestellt, die in jede Ressourcengruppe eingebunden werden muss. Nach der Registrierung können die gewünschten PrivacySettings über die Verwaltungsoberfläche der PMP festgelegt werden.

Eine Ressource kann dann an jeder beliebigen Stelle ihres Codes abfragen, ob ein bestimmtes PrivacySetting gewährt wurde. Damit werden die Zuordnung von PrivacySettings zu Ressourcen sowie die Behandlung nicht gewährter Berechtigungen zur Laufzeit vollständig dem Entwickler der jeweiligen Ressource überlassen.

Da Ressourcen über den PMP-AppService angesprochen werden, müssen sowohl die Ressourcengruppe als auch die aufrufende App eine identische *.aidl*-Datei besitzen, die die verfügbaren Interface-Methoden und deren Parameter definiert (vgl. Abschnitt 4.1). Darüber hinaus muss die App ebenfalls die *PMP-API* einbinden.

Die App kann mit *isServiceFeatureEnabled()* zunächst überprüfen, ob Berechtigungen für die Verwendung der jeweiligen Komponente gewährt wurden, und danach mit *getResource()* einen passenden *PMPRequestResourceHandler()* erzeugen, über den die im AIDL-Interface definierten Methoden der Ressource angesprochen werden können.

Abbildung 5.2 zeigt schematisch die von PMP-Apps und Ressourcen unbedingt benötigten Komponenten sowie deren Beziehung untereinander und zur PMP:

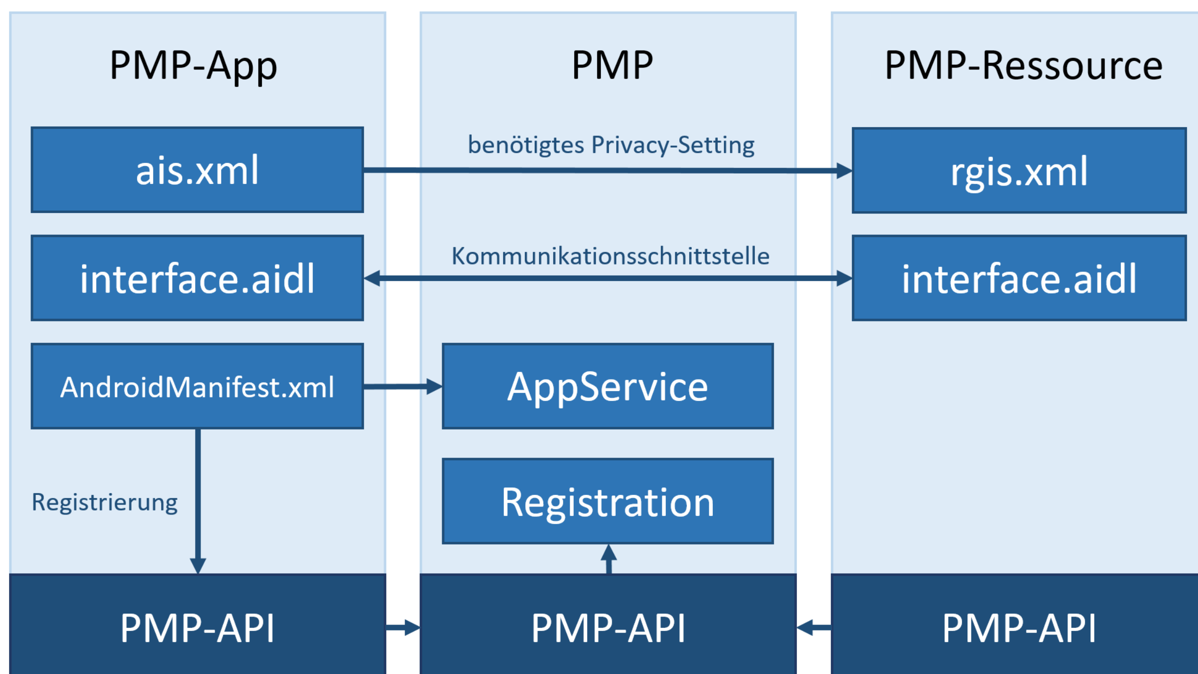


Abbildung 5.2: Technischer Aufbau der Privacy Management Platform

5.2 Implementierung des Protoyps

Bei der Implementierung des SOC auf Basis des vorhandenen PMP-Frameworks kann dessen bereits vorhandene Struktur ausgenutzt werden: Der SOC selbst wird hierbei als PMP-Ressource implementiert; von Apps benötigte Zugriffmethoden können dagegen als Teil der *PMP-API* abgelegt werden, da diese von allen PMP-Apps eingebunden wird.

5.2.1 Ressource

Wie in Abschnitt 5.1.2 beschrieben, werden Ressourcengruppen als Apps implementiert. Für den Protoyp reicht es, wenn das *resourceGroupInformationSet* nur ein einziges PrivacySetting *(„useSOC")* definiert. Die von der SOC-Ressource zur Abbildung der in Abschnitt 3.1.1 beschriebenen Schnittstellen benötigten Methoden werden innerhalb einer AIDL-Datei deklariert:

Listing 5.5 Interface der SOC-Ressource

```
package de.unistuttgart.ipvs.pmp.resourcegroups.soc;

interface SOC
{
   int putObject(in String key, in List<String> data);
   int getObject(in String key, out List<String> data);
   int deleteObject(in String key);
   int shareObject(in String key, in String app);
   int unshareObject(in String key);
   int register();
   int deleteAll();
}
```

Hierbei ist anzumerken, dass *putObject()* und *getObject()* für die SOC-interne Weitergabe (zwischen API und Ressource) jeweils vier Parameter haben sollten: Neben dem Schlüssel müssen die serialisierten Objektdaten sowie Angaben zu Klasse und Serialisierungsmethode übertragen werden; letztere drei sind bei *getObject()* jedoch Rückgabewerte.

Während die individuelle Angabe dieser Parameter bei *putObject()* kein Problem darstellt, ist es bei *getObject()* nicht möglich, da AIDL grundsätzlich nicht erlaubt, Strings als *out*-Parameter zu deklarieren [dre17]. Daher müssen die drei Werte zur Rückgabe in einer Liste oder einem Array gekapselt werden; die Liste wurde hierbei gewählt, damit die Schnittstelle bei Bedarf durch weitere Metadaten erweitert werden kann. Aus Konsistenzgründen wurde dieselbe Darstellungsform auch für *putObject()* verwendet.

Der Methoden-Rückgabewert *int* aller Schnittstellen wird als Fehlerprotokoll genutzt, wobei Null einer erfolgreichen Operation entspricht. Die möglichen Fehlerfälle werden hierbei zentral in der API-Bibliothek definiert, da sowohl Ressourcen als auch aufrufende Apps auf diese Zugriff haben (siehe Abschnitt 5.2.2).

Auf Basis der AIDL-Datei können mit *extends SOC.Stub* passende Klassengerüste automatisch generiert werden. Eine Ressource muss hierbei drei Varianten implementieren: *Normal*, *Cloak* und *Mock*. Letztere beiden können genutzt werden, um veränderte oder zufällige Daten zurückzugeben; im Rahmen dieses Prototyps werden hierbei leere Datensätze übermittelt.

Für die normale Implementierung des Interfaces wurden eine Reihe von Hilfsmethoden angelegt, um den Status der aufrufenden App sowie deren Zugriffsberechtigungen auf angeforderte Datensätze zu überprüfen:

- **isOwner (key, app)** - Aufrufende App ist Inhaber des Objekts
- **hasShare (key, app)** - Aufrufende App hat Zugriff auf das Objekt
- **isRegistered (app)** - Aufrufende App ist beim Container registriert
- **registerApp (app)** - Registrierung der aufrufenden App
- **encrypt (data, app)** - Verschlüsselung eines Datensatzes
- **decrypt (data, app)** - Entschlüsselung eines Datensatzes

Für Datenbankzugriffe bietet Android die Klasse *SQLiteDatabase*, die unter anderem Convenience-Methoden zum Lesen und Schreiben von Daten bereitstellt. Die Datenbank selbst wird über eine Instanz von *SQLiteOpenHelper* verwaltet und initialisiert. Hierbei wird *PRAGMA secure_delete = true* verwendet, um gelöschte Daten mit Nullen zu überschreiben (siehe Abschnitt 4.4.4).

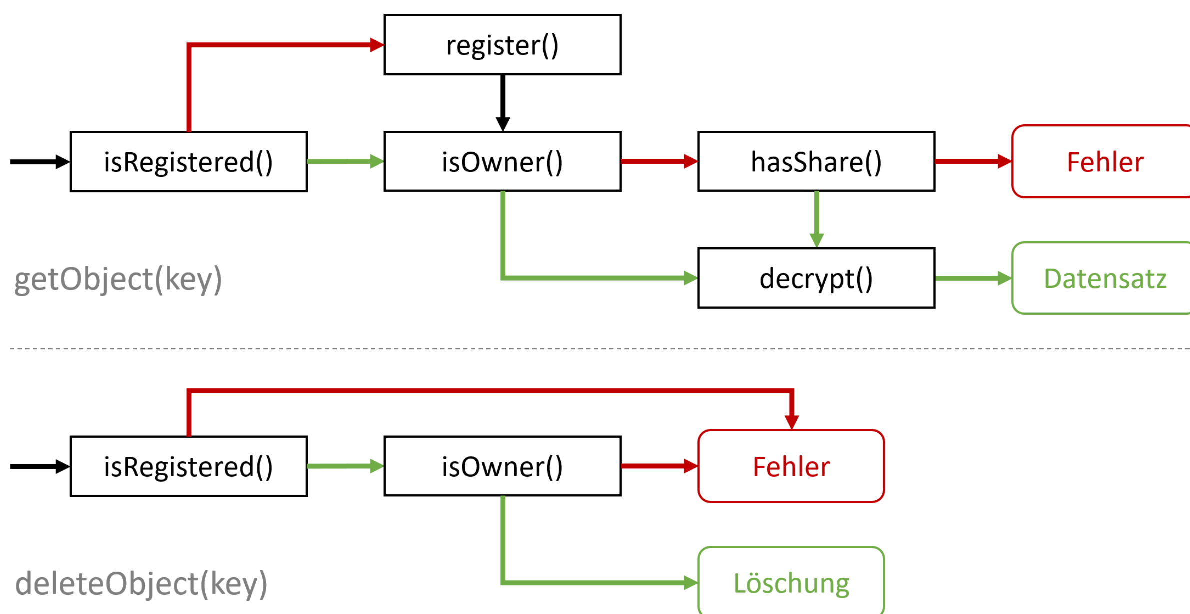


Abbildung 5.3: Methodenaufrufe zum Auslesen und Löschen von Datensätzen

Wie in Abschnitt 3.3 beschrieben, wird die jeweilige Operation zum Verschlüsseln bzw. Entschlüsseln erst aufgerufen, wenn alle Berechtigungschecks erfolgreich waren (siehe Abbildung 5.3). Der Prototyp verwendet das Verfahren *AES-256*, wobei jede App einen eigenen Schlüssel erhält, der bei Registrierung über einen *KeyGenerator* erzeugt wird. Bei Löschung der Daten einer App wird der dazugehörige Schlüssel ebenfalls entfernt.

Die einzig sichere Variante, die Schlüssel abzulegen, ist über einen *KeyStore* (siehe Abschnitt 4.4.3). Obwohl dieser in Android seit API-Level 18 verfügbar ist, werden symmetrische Schlüssel, wie sie für den SOC benötigt werden, erst ab API-Level 23 (Android 6.0) unterstützt. Da auf dem bereitgestellten Referenzgerät nur Android 5.1.1 läuft, könnte diese Variante daher nicht implementiert werden.

Stattdessen wird der Schlüssel exemplarisch als *SharedPreference* gespeichert (Dieser Ansatz bietet natürlich keinen ausreichenden Schutz für den realen Gebrauch). Die entsprechenden Methoden wurden in einer separaten Klasse *SOCencryption* gekapselt, die bei Bedarf durch eine KeyStore-Implementierung ausgetauscht werden kann. Khan beschreibt, wie eine symmetrische Verschlüsselung auf Basis von AES für Android 6.0+ aussehen könnte [Kha16]. Der dort außerdem vorgeschlagene Workaround für ältere Geräte funktioniert jedoch nicht, da die dafür benötigte Klasse *KeyProperties* ebenfalls erst seit API-Level 23 existiert.

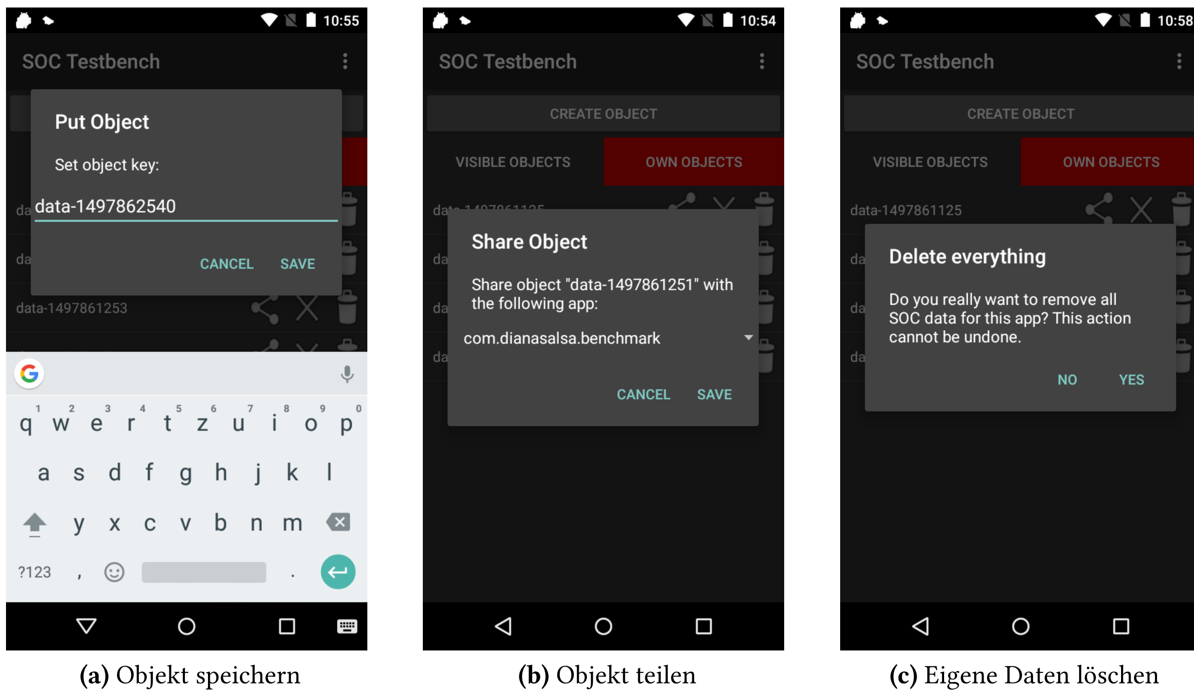
5.2.2 PMP-API

Um Zugriffe auf den SOC seitens der Apps zu vereinfachen und damit den Entwicklungsaufwand zu reduzieren (siehe Abschnitt 4.5), wurden Abbildungen der Zugriffsmethoden als Teil der PMP-API integriert. Diese unterteilen sich in drei separate Klassen:

- **SOCmanager** - Methoden für direkten Zugriff auf die SOC-Ressource
- **SOCmanagerUI**- Visuelle Elemente, Dialog-Wrapper für Zugriffsmethoden
- **SOCinterface** - Interface zur Rückgabe von Daten an die aufrufende Activity

Der *SOCmanager* bietet analog zu jedem Interface der SOC-Ressoure eine eigene Zugriffsmethode. Während der SOC die Daten jedoch in bereits serialisierter Form entgegennimmt, kann ein Objekt hier direkt übergeben werden. Serialisierung des Objekts sowie Verbindung mit der SOC-Ressource und Fehlerbehandlung werden alle automatisch innerhalb des SOCmanagers abgehandelt. Die aufrufende App muss daher lediglich *SOCmanager.putObject(key, object)* aufrufen. Gleiches gilt für alle abgebildeten Interface-Methoden (siehe Abschnitt 3.1.1).

In der *SOCmanagerUI* werden Komponenten zur visuellen Interaktion mit dem SOC bereitgestellt. Neben der Anzeige von Fehlermeldungen und sonstigem Feedback werden dort Methoden angeboten, die die SOCmanager-Aufrufe in Dialoge kapseln, etwa zur Eingabe von Daten oder Bestätigung von Löschoperationen (siehe Abbildung 5.4).



(a) Objekt speichern **(b)** Objekt teilen **(c)** Eigene Daten löschen

Abbildung 5.4: Dialog-Wrapper der SOCmanagerUI

Um eine Liste der verfügbaren Datensätze abrufen zu können, wurde die SOC-Ressource durch die Interfaces *getObjects*() und *getMyObjects()* ergänzt. Dies erlaubt beispielsweise, dem Anwender eine Auswahlliste anzuzeigen, wobei einzelne Einträge wiederum mit Funktionen des SOC verknüpft werden können (z.B. bearbeiten, löschen, teilen).

Da Service-Aufrufe grundsätzlich asynchron stattfinden, können vom SOC erhaltene Daten nicht direkt an die aufrufende App zurückgegeben werden. Hierzu wird eine weitere Kommunikationsschnittstelle benötigt, die innerhalb der API als *SOCinterface* definiert wurde. Eine Activity muss dafür die in Listing 5.6 gezeigten Methoden implementieren.

Listing 5.6 Interface zur Rückgabe der SOC-Daten

```
public interface SOCinterface
{
   public void receiveObject(Object object);
   public void receiveObjectList(List<String> list);
   public void receiveOwnObjectList(List<String> list);
   public void dataChanged(boolean reload);
}
```

Die SOC-API kann generell von jeder beliebigen Activity einer PMP-App aufgerufen werden; eine Implementierung des *SOCinterface* ist nur notwendig, sofern tatsächlich Daten empfangen werden sollen (Speichern, Löschen und Teilen ist immer möglich).

5.2.3 Gesamtübersicht

Basierend auf den in Abschnitt 5.1 vorgestellten Komponenten der PMP und der in Abschnitt 5.2 vorgestellten Implementierung von SOC-Ressource und API ergibt sich das in Abbildung 5.5 gezeigte Gesamtbild für die PMP-basierte Zusammensetzung des SOC-Prototyps.

Die grünen Pfeile illustrieren hierbei die Zugriffsreihenfolge bei Aufruf durch die App, die roten Pfeile beschreiben den Rückgabeverlauf nach Abruf von Daten. Hierbei sind jeweils mehrere Wege möglich, abhängig davon, ob Zugriffe über UI-Wrapper oder direkt geschehen und ob Objekte übertragen werden. Bei Rückgabe wird der Fehlercode ausgewertet und ggf. eine entsprechende Meldungen als Dialog- oder Benachrichtigung angezeigt. Für Datenrückgaben verwendet die API die SOCinterface-Implementierung der aufrufenden Activity.

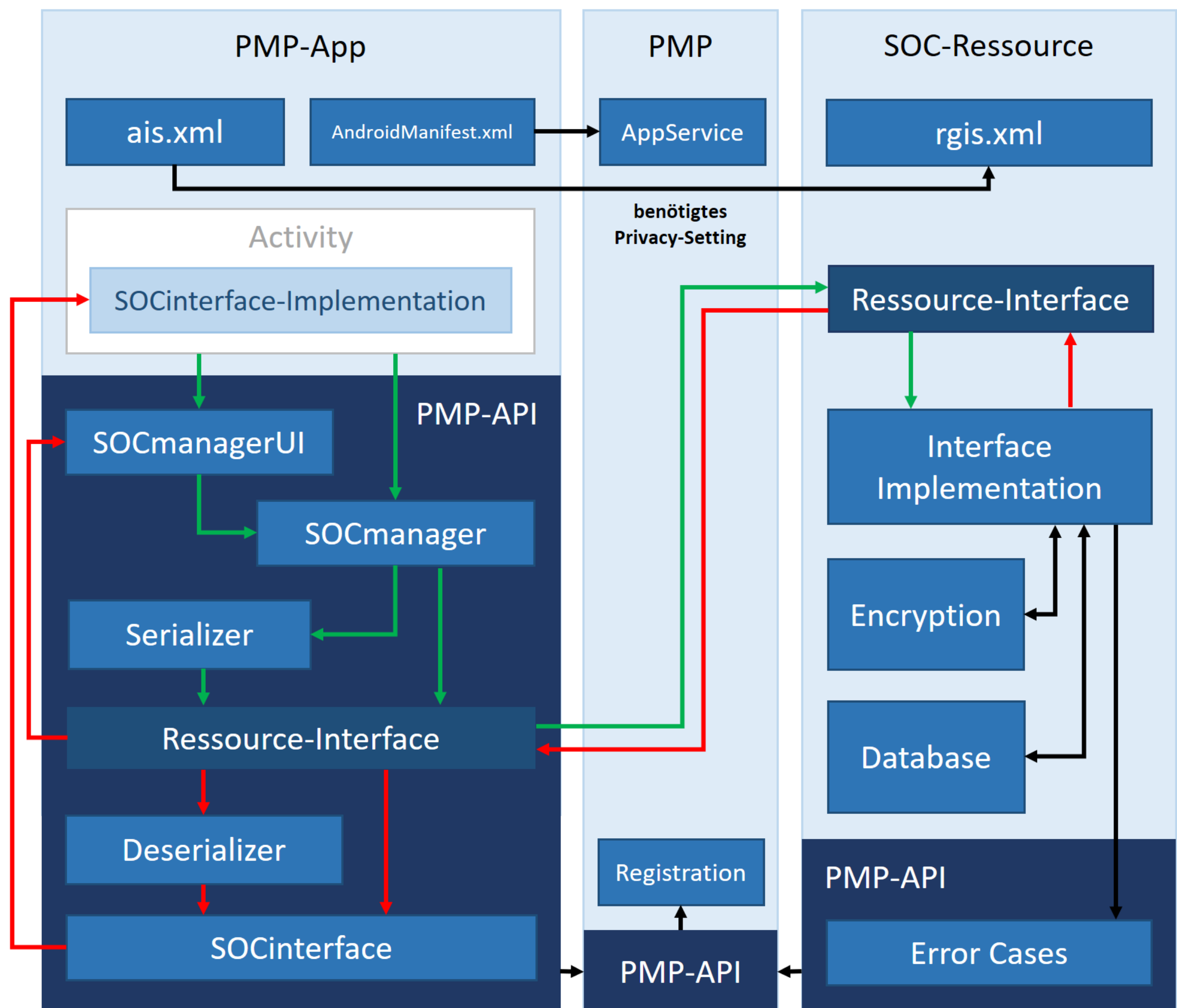


Abbildung 5.5: Kernkomponenten und Zugriffspunkte des SOC-Prototyps

6 Evaluation

Im Folgenden werden Funktionsweise und Performance des SOC mit bestehenden Ansätzen verglichen. Hierzu wurden zunächst die Geschwindigkeiten verschiedener Serialisierungsmethoden gemessen, um zu entscheiden, welche für den SOC am besten geeignet ist. Die daraus resultierende Variante des SOC wurde dann *Content Provider* und *SDC* gegenübergestellt.

Verwendet wurden hierbei Testgeräte aus drei verschiedenen Leistungsklassen: LG G4 (Android 6.0), Motorola Moto E2 (Android 5.1.1) und Asus Zenfone Max (Android 5.0.1), wobei das G4 das leistungsstärkste, das Zenfone das insgesamt leistungsschwächste Gerät darstellt.

6.1 Serialisierung

Zum Vergleich der Serialisierungsansätze erzeugte eine Testapp 10000 individuelle Objekte, die dann mit jedem der in Abschnitt 4.2 vorgestellten Verfahren serialisiert sowie deserialisiert wurden; dabei wurde die jeweils benötigte Zeit gemessen.

Es wurden zwei Varianten des Tests durchgeführt, einmal mit Objekten minimaler Klassenstruktur (zwei Felder) und einmal mit Instanzen der Klasse *MedicalData*, die wiederum ein eingebettetes Objekt der Klasse *Patient* enthalten (siehe Abbildung 6.1). Abbildung 6.2 und 6.3 zeigen die dabei gemessenen Durchschnittszeiten pro Objekt.

Aus dem Messergebnis ist klar ersichtlich, dass die von Java bzw. Android bereitgestellten Standardverfahren *Serializable*, *Externalizable* und *Parcelable* eine deutlich bessere Geschwindigkeit aufweisen als externe Bibliotheken. Der Vergleich zwischen beiden Testvarianten zeigt außerdem, dass dieser Unterschied mit steigender Komplexität der Objekte stark zunimmt. *Parcelable* schneidet insgesamt am besten ab, ist jedoch als Serialisierungsmethode für persistente Datenspeicher ungeeignet (siehe Abschnitt 4.2.1).

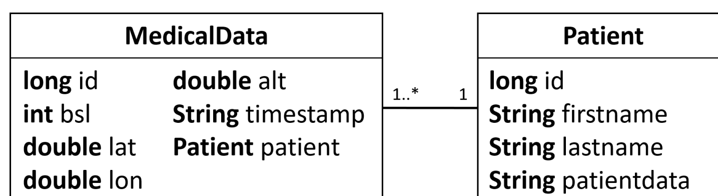


Abbildung 6.1: Aufbau der Testklassen MedicalData und Patient

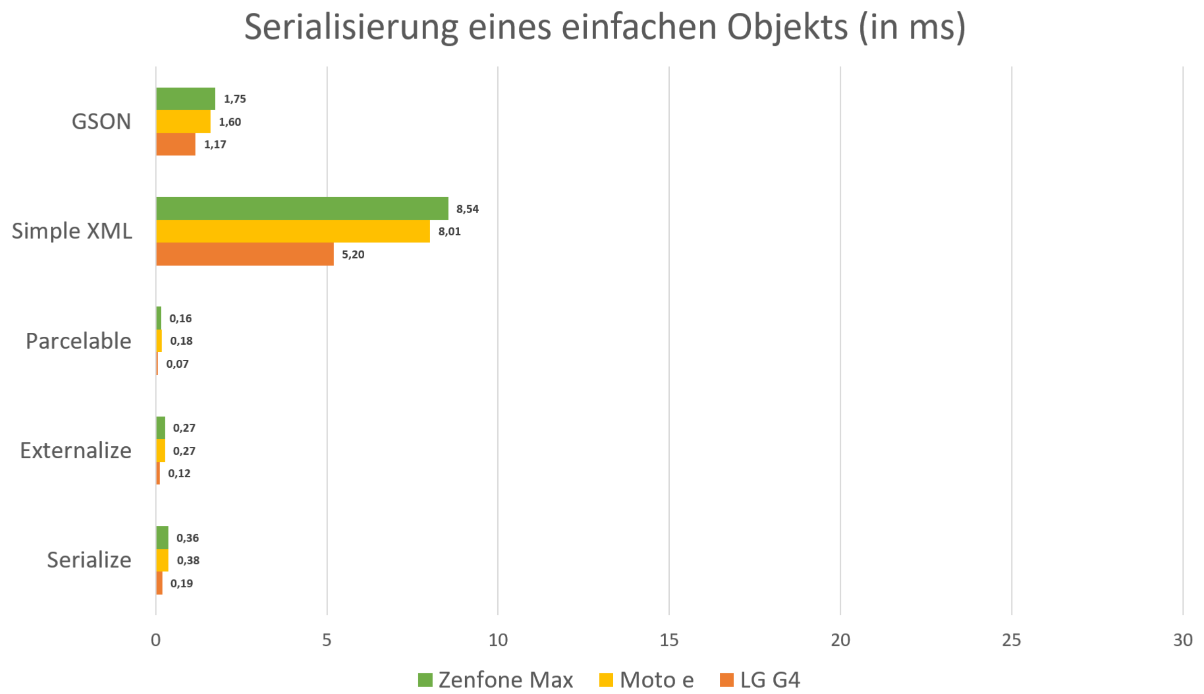


Abbildung 6.2: Serialisierungsgeschwindigkeit eines einfachen Objekts

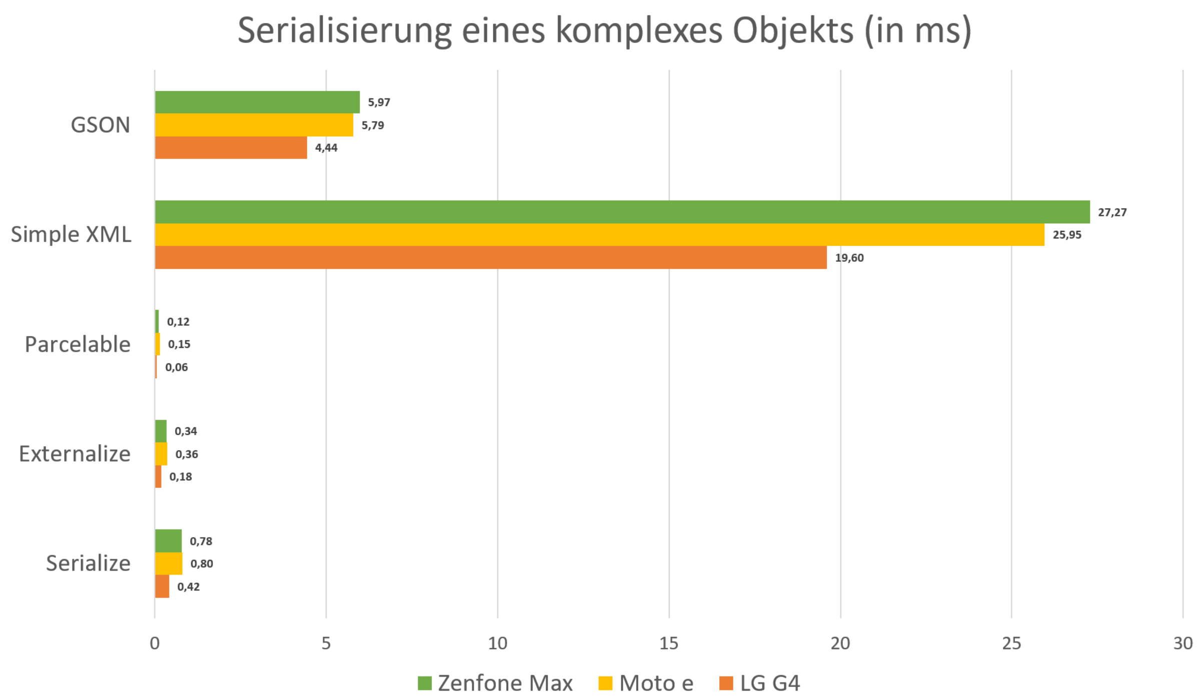


Abbildung 6.3: Serialisierungsgeschwindigkeit eines komplexen Objekts

Betrachtet man die implementierungsunabhängigen Verfahren, zeigt sich, dass GSON deutlich besser abschneidet als Simple XML. Dies könnte auf die Performance des verwendeten XML-Parsers zurückzuführen sein. Simple XML wird darüber hinaus seit 2013 nicht mehr aktiv gepflegt; insgesamt erscheint GSON daher deutlich attraktiver.

Vergleicht man auf Basis der verwendeten Hardware, sieht man bei den nativen Methoden nur Unterschiede von Bruchteilen einer Millisekunde. Auch GSON schneidet diesbezüglich noch relativ gut ab, bei XML sind dagegen auch sehr deutliche Unterschiede zwischen verschiedenen Geräteklassen erkennbar.

Zusätzlich zur Geschwindigkeit ist für die Nutzung innerhalb eines persistenten Datenspeichers auch die Größe relevant. Hierzu wurden nach der Serialisierung von Objekten der Klasse *MedicalData* deren jeweilige Größe gemessen. Abbildung 6.4 zeigt den direkten Vergleich zwischen den verschiedenen Verfahren.

Wenig überraschend ist, dass *Parcelable* und *Externalizable* eine deutlich kompaktere Darstellungsform erzeugen als *Serializable*. Interessant ist jedoch, dass GSON ebenfalls entsprechend wenig Platz benötigt, obwohl der Serialisierungsvorgang hierbei nicht durch eine manuelle Interface-Implementierung optimiert wurde. XML produziert im Vergleich deutlich größere Datensätze, was jedoch auf die Darstellungsform zurückzuführen ist (siehe Abschnitt 4.2.2).

Möchte man die Geschwindigkeit des SOC optimieren, ist demnach ein hybrider Ansatz sinnvoll; hierbei wird das Objekt zunächst auf Unterstützung von *Serializable* bzw. *Externalizable* geprüft und GSON lediglich als Fallback verwendet (siehe Abschnitt 4.2.3).

Da eine Implementierung nativer Serialisierungsverfahren bei beliebigen Klassen jedoch nicht vorausgesetzt werden kann, wurde zur folgenden Vergleichsmessung GSON als alleinige Lösungsstrategie für den SOC eingesetzt.

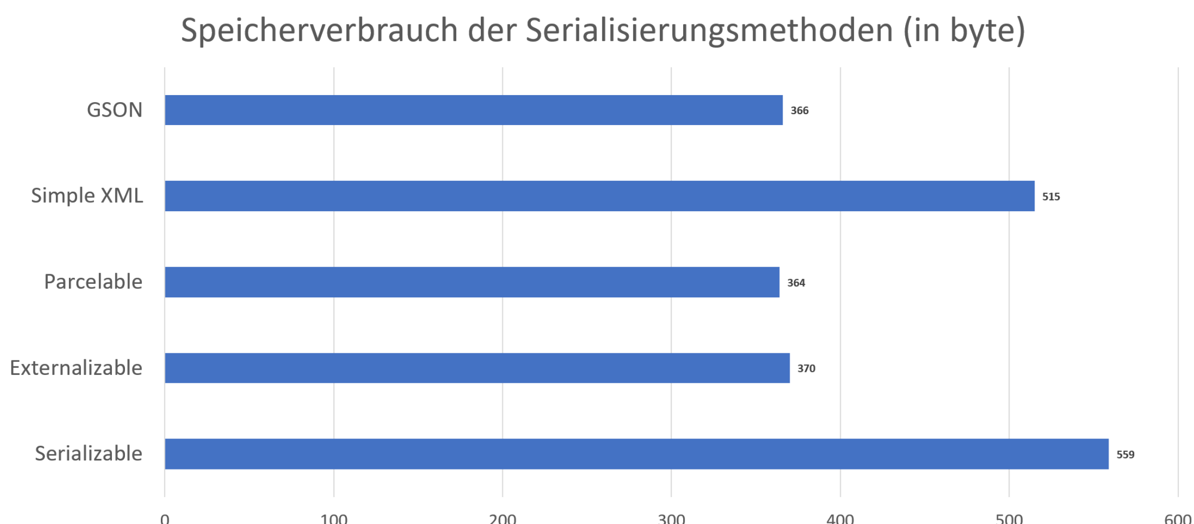


Abbildung 6.4: Speicherverbrauch der verschiedenen Serialisierungsmethoden

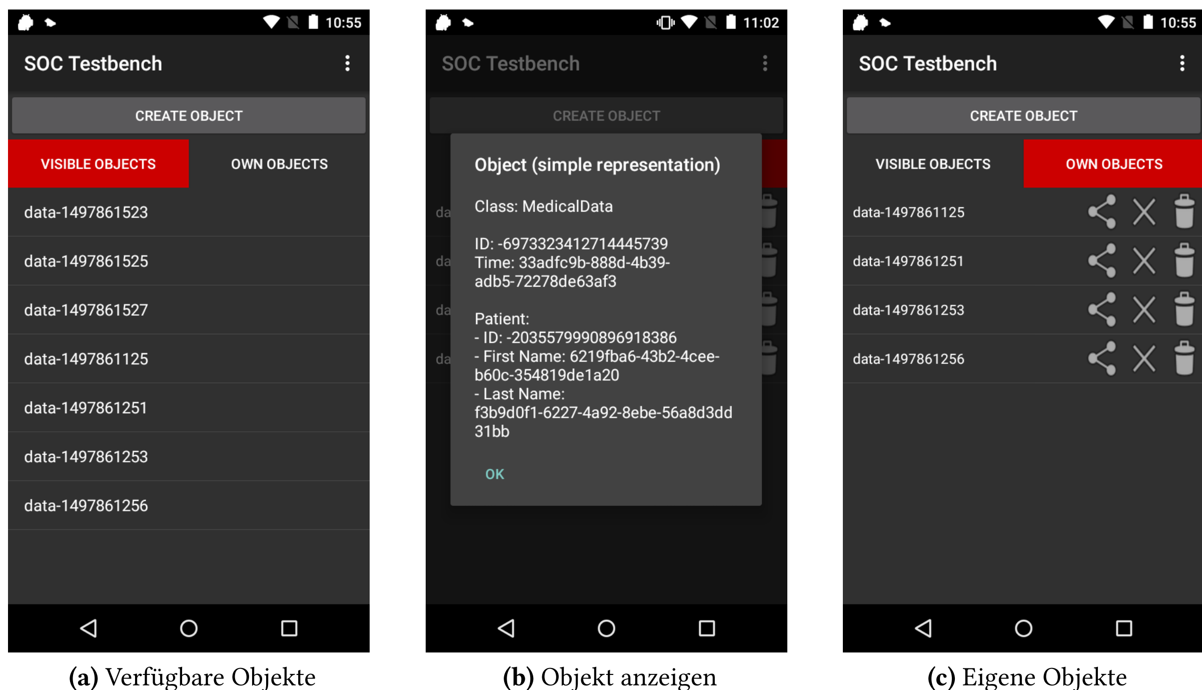
6.2 SOC-Testapp

Um zu zeigen, dass alle implementierten Funktionen des SOC erwartungsgemäß funktionieren, wurde eine App auf Basis der API implementiert, die alle Schnittstellen abbildet (siehe Abbildung 6.5). Als Beispielobjekt wird auch hier die Klasse *MedicalData* verwendet.

Bei Erzeugung jedes neuen Objekts werden dessen Felder mit zufälligen Werten befüllt; bei eigenen Objekten besteht die Möglichkeit, diese mit anderen installierten Apps zu teilen oder zu löschen. Alle Objekte werden nach ihrem Schlüssel sortiert aufgelistet, ein Klick darauf ruft den entsprechenden Datensatz vom SOC ab. Das Löschen aller eigenen im SOC abgelegten Daten sowie die erstmalige Registrierung sind über das Menü möglich.

Der manuelle Umgang mit der App zeigt, dass Operationen in einem für Anwender akzeptablen Zeitfenster ausgeführt werden; es entsteht keine merkliche Latenz beim Speichern und Abrufen einzelner Objekte und Freigaben.

Die App wurde mithilfe der in Listing 5.6 definierten Kommunikationsschnittstellen so eingerichtet, dass sich bei jedem Hinzufügen und Löschen eines Objekts die Anzeige aktualisiert, also die gesamte Objektliste neu vom SOC angefordert wird. Hierbei wird die Latenz erst bei einer Anzahl von hunderten bis tausenden Objekten überhaupt bemerkbar.



(a) Verfügbare Objekte **(b)** Objekt anzeigen **(c)** Eigene Objekte

Abbildung 6.5: Benutzeroberfläche der SOC-Testapp

6.3 Geschwindigkeit

Im Folgenden wird analysiert, wie schnell der SOC-Prototyp im Vergleich mit nativem Content Provider (siehe Abschnitt 2.1.3) und Secure Data Container (siehe Abschnitt 2.2.4) ist. Hierzu werden SOC-Ressource und -API durch folgende Methoden ergänzt:

- **benchmarkWrite(keys, objects)** - Serialisierung der Objekte + Speichern im SOC
- **benchmarkRead(keys)** - Lesen der Datensätze aus dem SOC + Deserialisierung

Die SOC-Testapp wurde dafür um eine zusätzliche Activity erweitert, über die eine beliebig wählbare Anzahl von Objekten und Schlüssel zufällig erzeugt und die beiden Benchmark-Methoden aufgerufen werden können. Hierbei muss zunächst die Schreib-Benchmark ausgeführt werden, die die Objekte im SOC anlegt, so dass sie danach von der Lese-Benchmark wieder abgerufen werden können.

Als Vergleichsbasis dienen analoge Benchmarks für SDC und Content Provider. Alle Tests verwenden dabei zufällig erzeugte Dummy-Objekte derselben Klasse (*MedicalData*), bei jedem Test wurden drei Messungen mit je 1000, 5000 und 10000 Objekten auf den verschiedenen Geräten durchgeführt.

Die Abbildungen 6.6 und 6.7 zeigen die dabei insgesamt (auf Basis aller Messungen) ermittelten Durchschnittswerte für die Dauer einer Lese- und Schreiboperation. Hierbei ist erkennbar, dass der SOC beide Operationen deutlich schneller ausführt als SDC und Content Provider.

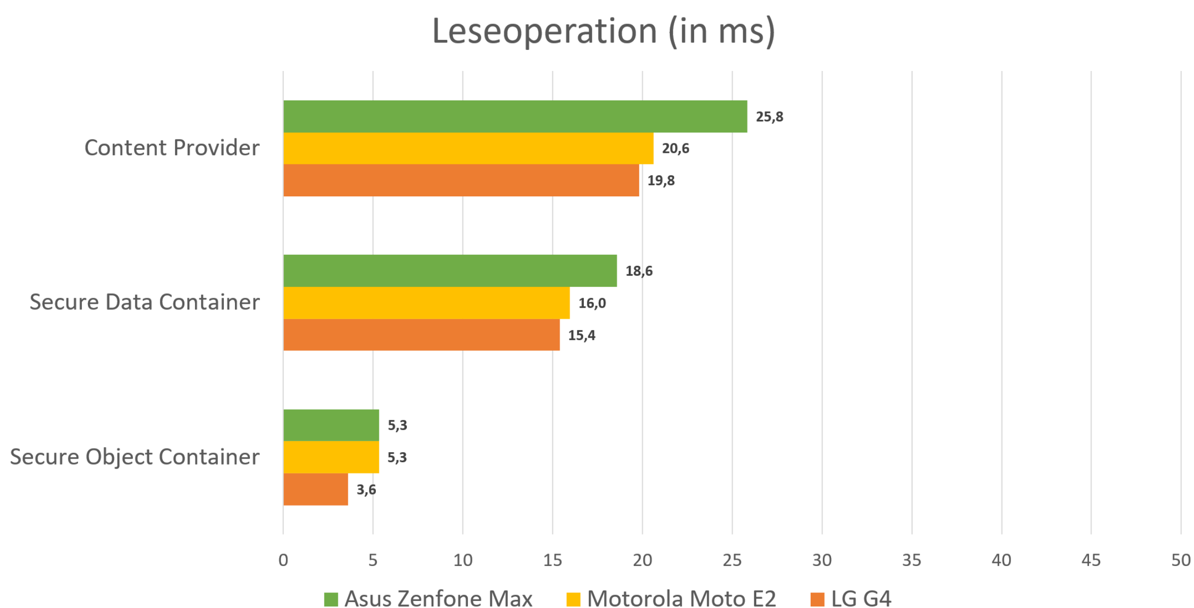


Abbildung 6.6: Vergleich: Durchschnittliche Dauer einer Leseoperation

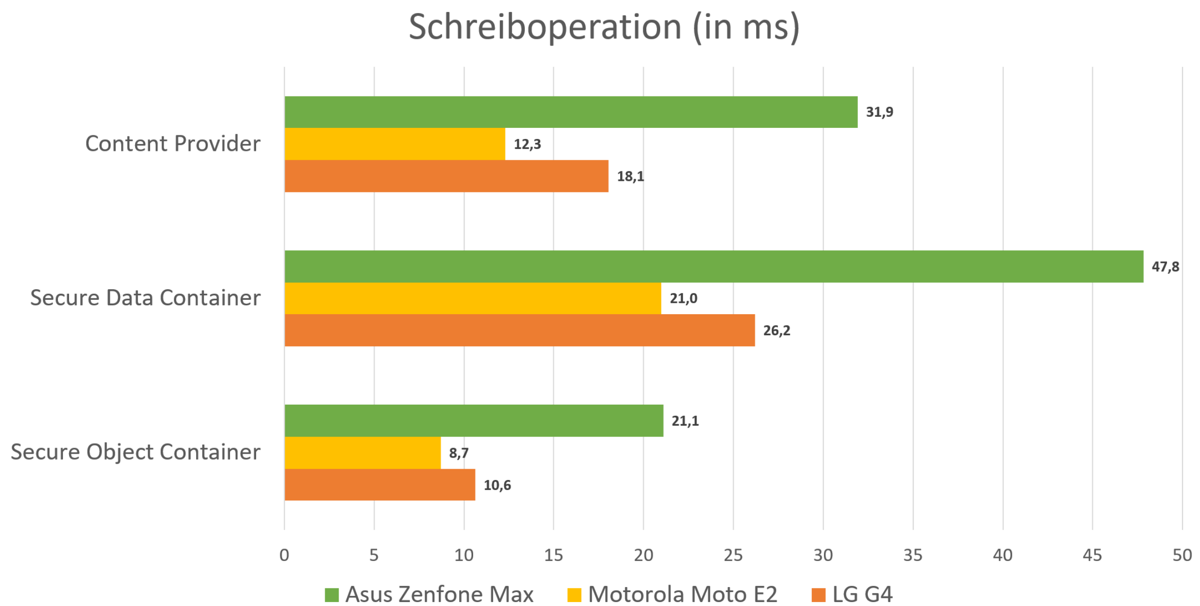


Abbildung 6.7: Vergleich: Durchschnittliche Dauer einer Schreiboperation

Überraschend ist dies nicht, da die beiden anderen Verfahren Objekte vor dem Ablegen in der Datenbank manuell in ihre Felder zerlegen und auf verschiedene Tabellen verteilen müssen (genauso umgekehrt). Beim SOC werden die Objekte dagegen serialisiert und als einzelner Wert gespeichert. Insgesamt ist daher erkennbar, dass der Serialisierungs-Overhead deutlich geringer ist als die manuelle Zerlegung auf Datenbankebene.

Interessant ist dagegen, dass das LG G4, obwohl insgesamt das leistungsstärkste Gerät, deutlich schlechtere Schreibgeschwindigkeiten aufweist als das Moto E2. Erklären lässt sich dies durch die generell unterdurchschnittliche SQLite-Schreibgeschwindigkeit des G4. Tabelle 6.1 zeigt eine Übersicht der drei Testgeräte bezüglich ihrer SQLite-Performance:

	SQLite read	SQLite update	SQLite insert	SQLite delete
LG G4	2323 IOPS	1000 IOPS	80 IOPS	333 IOPS
Motorola Moto e	1404 IOPS	267 IOPS	122 IOPS	222 IOPS
Zenfone Max	1736 IOPS	90 IOPS	49 IOPS	69 IOPS

Tabelle 6.1: SQLite-Performance der verwendeten Testgeräte, gemessen mit 3DMark (Input/Output-Operationen pro Sekunde) [fut17a; fut17b; fut17c; fut17d]

Obwohl im Test nur GSON zur Serialisierung genutzt wurde (welches deutlich langsamer ist als native Methoden – siehe Abschnitt 6.1), schneidet der SOC überraschend gut ab. Es wäre daher sogar denkbar, dieses Verfahren als alleinige Serialisierungsmethode zu implementieren; dadurch würde der Overhead gespart, der durch das Auswählen, Speichern und Abfragen der verwendeten Serialisierungsmethode entsteht.

6.4 Speicherverbrauch

Um den Speicherverbrauch aller drei Verfahren zu vergleichen, wurden neue Instanzen der jeweiligen Datenbanken initialisiert und wie in Abschnitt 6.3 Schreiboperationen mit *benchmarkWrite()* durchgeführt, wobei nach jeder Objektgruppe die Größe der Datenbank gemessen wurde. Verwendet wurden die Maßzahlen *PRAGMA schema.page_count* und *PRAGMA schema.page_size*, deren Produkt den Speicherverbrauch ergibt [sql17].

Abzüglich der Initialgröße der leeren Datenbank wurde aus den gemessenen Werten der durchschnittliche Speicherverbrauch eines Objekts der Klasse *MedicalData* für das jeweilige Verfahren ermittelt. Abbildung 6.8 zeigt die Unterschiede im direkten Vergleich; bei allen drei Testgeräten wurden dabei nahezu identische Werte gemessen, daher wurde hier auf deren Unterscheidung verzichtet.

Das Ergebnis zeigt deutlich die Vorzüge relationaler Datenbanken gegenüber serialisierten Datenstrukturen. Bei der Serialisierung werden nicht nur die Werte der einzelnen Felder, sondern auch deren Bezeichner gespeichert (Dies gilt konkret für GSON und XML; die Messung in Abschnitt 6.1 zeigt jedoch, dass auch binäre Serialisierung nicht weniger Platz verbraucht). Relationale Datenbanken speichern dagegen nur die Werte jedes Datensatzes, da die dazugehörigen Felder global über das Schema definiert sind.

Lediglich die initiale Datenbankgröße ist beim SOC kleiner, was darauf zurückzuführen ist, dass aufgrund des reduzierten Schemas weniger Tabellenspalten angelegt werden müssen. Im Verhältnis zur Gesamtmenge der gespeicherten Daten fällt dieser Unterschied jedoch praktisch nicht ins Gewicht.

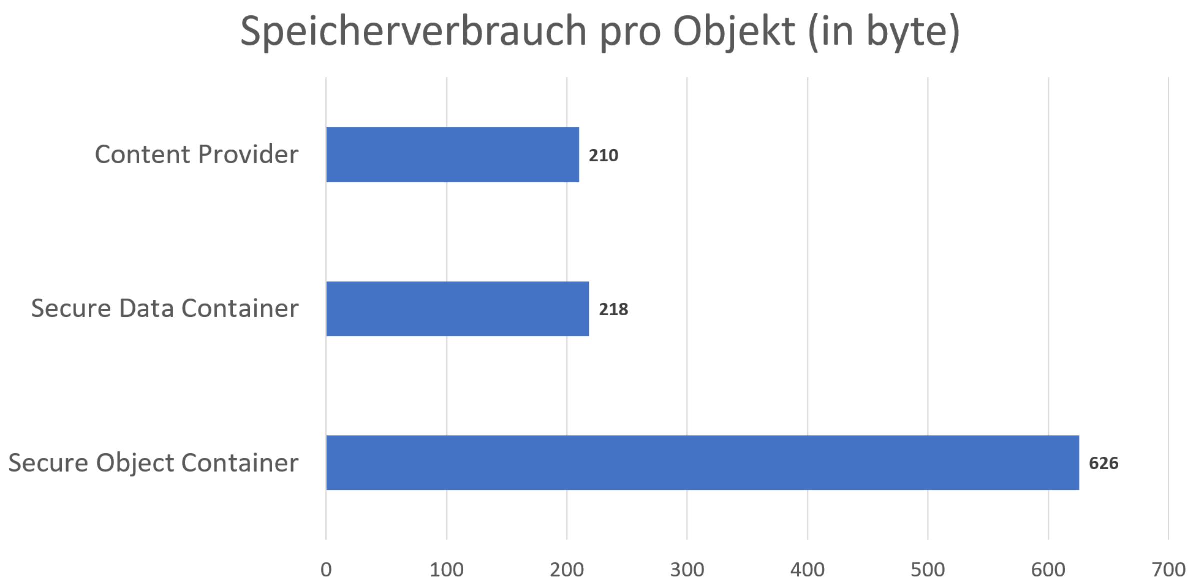


Abbildung 6.8: Vergleich: Durchschnittlicher Speicherverbrauch pro Objekt

6.5 Erfüllung der Anforderungen

Im Folgenden wird bewertet, inwieweit der SOC die in Abschnitt 1.3 definierten Anforderungen erfüllt. Hierbei wird die „korrekte“ Implementierungsvariante auf Basis eines Android KeyStores mit Hardwareunterstützung zu Grunde gelegt, da dieser die Voraussetzung für sämtliche Sicherheitsgarantien darstellt (siehe Abschnit 4.4.3).

Confidentiality (Vertraulichkeit) - Alle Datensätze werden im internen Dateisystem des SOC gespeichert, Zugriff ist nur über die bereitgestellten Schnittstellen möglich. Bei jedem Zugriff wird die Identität der aufrufenden App überprüft, nur der Besitzer des Datensatzes sowie Apps, die eine Freigabe des Inhabers erhalten haben, können darauf zugreifen. Ein Angreifer, der direkten Zugriff auf die interne Datenbank erlangt, kann die Objektdaten nicht auslesen, da diese verschlüsselt gespeichert sind. Der dazugehörige Schlüssel liegt in einem durch sichere Hardware geschützten KeyStore und kann nur vom SOC verwendet werden.

Integrity (Unversehrtheit) - Analog zum vorherigen Punkt kann nur der Besitzer eines Datensatzes diesen verändern. Dies gilt sowohl für gespeicherte Objekte als auch für dazugehörige Freigaben. Eine Manipulation der gespeicherten Objektdaten durch einen Angreifer mit Schreibrechten auf die Datenbank kann lediglich zur Löschung/Zerstörung eines Datensatzes führen; eine unentdeckte Änderung ist nicht möglich, da der Angreifer den zum Speichern benötigten Schlüssel nicht kennt. Würde der Angreifer stattdessen das owner-Feld eines Objekts zu seinen Gunsten manipulieren, könnte er die gespeicherten Daten dennoch nicht lesen, da diese nicht zu seinem Schlüssel passen. Das Anlegen einer neuen Freigabe ist ebenfalls nicht möglich, da hierzu ebenfalls der Schlüssel des Besitzers benötigt wird (siehe Abschnitt 3.3).

Availability (Verfügbarkeit) - Der SOC ist als PMP-Ressource implementiert, die im Hintergrund als Service aktiv ist. Alle Daten werden persistent gespeichert, Zugriffe sind jederzeit über die bereitgestellten Schnittstellen möglich, solange das Gerät eingeschaltet und entsperrt ist. Letzteres ist bedingt durch die KeyStore-Implementierung, die Zugriff auf gespeicherte Schlüssel aus Sicherheitsgründen nur bei entsperrtem Gerät erlaubt (siehe Abschnitt 4.4.3). Operationen, die keinen Schlüsselzugriff benötigen (etwa das Löschen von Daten) könnten daher trotzdem durchgeführt werden.

Feingranulare Berechtigungen - Im SOC gespeicherte Datensätze kann zunächst nur deren Besitzer abrufen, einzelne Objekte können jedoch für weitere Apps freigegeben werden. Diese zusätzlichen Berechtigungen werden individuell gespeichert: Der Schlüssel eines Objekts zusammen mit dem Namen einer berechtigten App ergibt einen Freigabedatensatz. Bei Abruf eines Objekts wird die Zugriffsberechtigung für Nichtbesitzer auf Basis dieser Datensätze überprüft; existiert kein passender Eintrag, wird die Anfrage abgelehnt.

Genericity (Allgemeine Anwendbarkeit) - Der SOC erlaubt die Ablage beliebiger Objekte und stellt generische Methoden und Speicherstrukturen für deren Verwaltung bereit. Dabei werden keinerlei Anforderungen an die Klassenimplementierung der Objekte gestellt, da als Fallback eine implementierungsunabhängige Serialisierung mit GSON stattfindet.

Usability (Benutzerfreundlichkeit) - Entwickler einer App können für Zugriffe auf den SOC eine zentral gepflegte API-Bibliothek nutzen. Ablegen, Lesen, Freigeben und Löschen von Objekten kann direkt mit dafür bereitgestellten Convenience-Methoden durchgeführt werden, während die API sämtliche internen Vorgänge vor dem Entwickler verbirgt. Ebenfalls werden Elemente zur Bedienung über eine grafische Benutzeroberfläche angeboten.

Insgesamt ist klar erkennbar, dass der SOC alle im Vorfeld definierten Anforderungen erfüllt.

6.6 SOC vs. SDC

Secure Data Container (SDC) und *Secure Object Container (SOC)* stellen alternative Ansätze zur Implementierung einer sicheren Speicherplattform für den App-übergreifenden Austausch von Daten dar. Obwohl der SOC alle im Vorfeld definierten Anforderungen erfüllt, bedeutet dies nicht, dass er dem SDC in allen Anwendungsszenarien überlegen ist. Die beiden Ansätze werden daher abschließend bezüglich ihrer jeweiligen Vor- und Nachteile verglichen.

6.6.1 Vorteile des SOC

- **Geschwindigkeit** - Wie Abschnitt 6.3 deutlich zeigt, benötigt der SOC für gleichwertige Schreibzugriffe nur die Hälfte, für Lesezugriffe sogar nur ein Drittel der Zeit der SDC.
- **Generische Objekte** - Der SOC kann aufgrund seines generischen Datenbankschemas beliebige Objekte speichern. Der SDC muss dagegen jede unterstützte Klassenstruktur explizit in Tabellenform abbilden; die Änderungen vorhandener Strukturen sowie das Hinzufügen neuer Klassen erfordern daher eine Anpassung der Container-Implementierung.
- **Generischer Zugriff** - Beim SOC können dieselben Schnittstellen für jede Art von Objekttyp genutzt werden. Der SDC definiert dagegen individuelle Methoden für jeden unterstützten Objekttyp; hierbei werden Objekte durch ein *Parcelable*-Interface übertragen, was die Nutzung auf entsprechend implementierte Klassen einschränkt.

Der SOC-Ansatz ist damit für die Implementierung eines Datenspeichers geeignet, der den möglichst einfachen Austausch von Objekten beliebiger Apps gestatten soll, ohne dabei Einschränkungen bezüglich deren Struktur zu definieren. Gleichzeitig kann er Objekte jedoch nur auf Basis ihres Schlüssels anfordern, ähnlich wie ein Key/Value-Store.

Als allgemeine Plattform zum Datenaustausch kann er damit für alle Arten von Freigaben genutzt werden, bei denen durch die Abfrage keine Vorab-Filterung (aus Sicht der darauf zugreifenden App) stattfinden soll. Anwenden lässt sich dies unter anderem auf das in Abschnitt 1.2 erwähnte Beispiel der Adressbuch-Weitergabe an einen Messenger-Dienst. Ein solcher Dienst möchte grundsätzlich *alle* verfügbaren Datensätze überprüfen, demnach würde er in keinem Fall eine Filterung bei der Datenbank-Abfrage anwenden.

6.6.2 Vorteile des SDC

- **Speicherverbrauch** - Wie in Abschnitt 6.4 gezeigt, benötigt der SDC verglichen mit dem SOC nur etwa ein Drittel des Speicherplatzes für dieselbe Objektstruktur, was auf die platzsparende Abbildung auf relationale Schemata zurückzuführen ist.
- **Attribut-basierte Abfragen** - Durch die relationale Abbildung der einzelnen Felder jedes Objekts können diese bei Datenbankzugriffen individuell angesprochen werden, was komplexe Abfragen auf Basis einzelner Feldwerte ermöglicht.

Während die Berücksichtigung des Speicherverbrauchs heute einen geringeren Stellenwert hat, ist die Möglichkeit selektiver Abfragen umso wichtiger, um Übertragungszeiten und damit schlussendlich auch die Geschwindigkeit einer Anwendung zu verbessern. Insbesondere bei der Speicherung sehr großer Datenmengen ist eine Vorab-Auswahl daher wichtig.

Im Gegensatz zum SOC kann der SDC jedoch nur als Lösung für eine begrenzte Gruppe von Apps eingesetzt werden, die Daten untereinander austauschen sollen. Hierbei muss ein einheitlicher Standard für Datensätze definiert werden, der sich sowohl in der Klassenimplementierung der Apps als auch in deren Abbildung innerhalb des Containers widerspiegelt.

Denkbar wäre dies beispielsweise bei einer Sammlung von Apps zur Messung und Auswertung von Gesundheitsdaten: Für jedes Gerät, das Gesundheitsinformationen überwacht (z.B. Bluetooth-Armband, Blutzucker-Messgerät oder Pulsfrequenzmesser) wird eine eigene App bereitgestellt, die die dazugehörigen Daten aufzeichnet. Durch Freigabe der Daten für eine zentrale App könnten diese dann in Form einer Gesamtübersicht aufbereitet werden.

7 Zusammenfassung und Ausblick

Mobilgeräte werden heutzutage in fast allen Bereichen des täglichen Lebens verwendet und speichern eine große Menge persönlicher Informationen. Anwender haben jedoch nur eingeschränkt Kontrolle darüber, in welcher Form diese Daten weiterverarbeitet werden; meist besteht lediglich die Alternative, einer App den Zugriff auf Daten vollständig zu verbieten, was i.d.R. mit entsprechenden Funktionseinbußen einhergeht. Viele Anwender fühlen sich von derartigen Entscheidungen überfordert, was häufig zu einer pauschalen Gewährung von Berechtigungsanfragen führt.

Die Weitergabe bestimmter Daten zwischen Apps kann für bestimmte Anwendungsfälle erforderlich und auch sinnvoll sein. Android stellt eine Reihe von Komponenten bereit, über die der Austausch nahezu beliebiger Datenstrukturen zwischen Apps möglich wird, jedoch wird hierbei der Schutz vor unberechtigten Zugriffen Dritter weitgehend vernachlässigt. Selbst über das traditionelle Berechtigungssystem definierte Freigaben können die darunterliegenden Daten nicht vollständig schützen; darüber hinaus besteht in den meisten Fällen keine Möglichkeit, Zugriffsrechte für individuelle Datensätze zu definieren.

Alternative Implementierungsansätze für Datenfreigaben zeigen, wie der Schutz von Daten in Zusammenhang mit feingranularem Berechtigungsmanagement realisiert werden kann. Keiner der vorgestellten Ansätze erfüllt jedoch alle Anforderungen, insbesondere kann eine umfassende Kontrolle aller Sicherheitsbestimmungen nur mit Änderungen am darunterliegenden Betriebssystem ermöglicht werden.

Der *Secure Data Container (SDC)* deckt einen Großteil der Anforderungen (insbesondere bezüglich der Sicherheit) ab, ist jedoch nicht für beliebige Datenstrukturen einsetzbar. *MetaService* erlaubt dagegen den Transfer aller Objekte, bietet jedoch keinerlei Sicherheitsgarantien. Der *Secure Object Container (SOC)* soll daher ein Konzept definieren, welches die Sicherheitsaspekte des SDC mit den generischen Schnittstellen des MetaService kombiniert.

Die daraus resultierende Plattform wurde als Prototyp für das Android-Betriebssystem realisiert. Hierzu werden zunächst Alternativen für die Implementierung einzelner Kernkomponenten verglichen und bewertet. Der SOC teilt sich dabei in zwei grundlegende Bereiche: eine API, die App-Entwicklern den möglichst einfachen Umgang mit dem SOC ermöglicht, und einen Service, der für Sicherheit, Datenverwaltung und Freigaben verantwortlich ist.

Analog zum SDC wurde auch der SOC-Prototyp als Teil der *Privacy Management Platform (PMP)* integriert. Hierbei handelt es sich um ein alternatives Berechtigungssystem für Android, bei dem Berechtigungen nicht Apps, sondern individuellen Funktionen zugewiesen werden. Die PMP stellt dabei Schnittstellen zur Kommunikation zwischen API und Service bereit.

Bei Evaluation des Prototyps werden zunächst Geschwindigkeit und Speicherverbrauch verschiedener Serialisierungsmethoden verglichen. Hierbei sind native Methoden am schnellsten, jedoch von der jeweiligen Klassenimplementierung abhängig. GSON zeigt als implementierungsunabhängige Lösung äquivalenten Speicherverbrauch und relativ gute Geschwindigkeit und wurde daher als Basis für die SOC-Testmessungen gewählt.

Dabei zeigte sich, dass der SOC im Vergleich mit SDC und Content Provider deutliche Verbesserungen der Zugriffsgeschwindigkeit bietet, jedoch insgesamt mehr Speicherplatz verbraucht. Insgesamt erfüllt der SOC alle Anforderungen, die SDC und MetaService summiert abdecken, wobei sein größter Vorteil in der Unterstützung beliebiger Datenobjekte liegt. Zuletzt wurden SOC und SDC direkt verglichen; hierbei wurde deutlich, dass es vom jeweiligen Anwendungsfall abhängt, welche Lösungsstrategie am besten geeignet ist.

Ausblick

Mit dem im Rahmen dieser Arbeit vorgestellten Secure Object Container ist das Speichern und Teilen beliebiger komplexer Objekte möglich, Freigaben werden individuellen Objekten und Apps zugeordnet. Schnittstellen und Speicherstruktur sind generisch, daher kann auf Objekte nur über deren Schlüssel zugegriffen werden. Im Folgenden werden zwei Ideen vorgestellt, wie diese Funktionalitäten erweitert werden könnten:

Freigabegruppen

Zunächst soll erneut der in Abschnitt 1.2 erwähnte Anwendungsfall der Weitergabe von Kontaktdaten an Messenger betrachtet werden. Im Gegensatz zum traditionellen Berechtigungsmanagement, welches Freigaben nur für die vollständige Liste vergibt, können über den SOC individuelle Adressen ausgewählt werden. Dabei müssen Daten, die zu einem späteren Zeitpunkt dem Adressbuch hinzugefügt wurden, daraufhin ebenfalls manuell geteilt werden.

Dies entspricht der sichersten Variante, um unerwünschte Zugriffe zu vermeiden, jedoch ist es nicht besonders benutzerfreundlich. Auf App-Seite könnte dieser Vorgang vereinfacht werden, indem der Anwender beispielsweise über Checkboxen einzelne Einträge markieren kann, die dann entsprechend geteilt oder zurückgezogen werden; jedoch erfordert dies trotzdem eine manuelle Freigabe bei jedem neuen Kontakt.

Da Kontakte in Gruppen gespeichert werden, wäre aus Anwendersicht eine entsprechende Abbildung auf die Freigabeverwaltung naheliegend. Dabei würden Kontakte, die einer geteilten Gruppe hinzugefügt werden, ebenfalls automatisch geteilt. Ebenso sollte der SOC dieselben Möglichkeiten wie traditionelle Systeme bieten, also die automatische Freigabe aller eigenen Kontakte, da auch dies in Einzelfällen sinnvoll sein kann (etwa für Mail- oder SMS-Apps).

Um dies SOC-seitig zu realisieren, müssten Datenbankschema und Schnittstellen dahingehend ergänzt werden, dass sie die Abbildung einzelner Datensätze auf Gruppen ermöglichen. Die Freigabeverwaltung könnte darauf aufbauend neben der Freigabe einzelner Datensätze auch die von Gruppen oder sogar aller Daten einer App unterstützen.

Geht man noch einen Schritt weiter, wäre sogar ein regelbasiertes Freigabeschema denkbar, nach der Form: *„Alle Datensätze vom Typ X, die Eigenschaft Y erfüllen, sollen automatisch mit App Z geteilt werden"*. Diese Art von Logik wäre jedoch ebenfalls App-seitig implementierbar; hierbei muss eine Abwägung getroffen werden, inwieweit sich ein solcher Ansatz mit der Anforderung verträgt, den SOC möglichst generisch zu halten.

Document Store

Der in Abschnitt 6.1 durchgeführte Vergleich von Serialisierungsmethoden hat ergeben, dass GSON eine in der Praxis durchaus verwendbare Alternative gegenüber nativen Verfahren ist. Zwar bieten diese eine höhere Geschwindigkeit, jedoch kann im Normalfall nicht davon ausgegangen werden, dass Klassen die benötigten Interfaces *Serializable* und *Externalizable* implementieren.

Würde GSON daher als alleinige Implementierungsstrategie verwendet, kann die Wahl des darunterliegenden Datenspeichers (siehe Abschnitt 4.3) neu bewertet werden. Aus der Familie der NoSQL-Datenbanken bieten Document Stores die Möglichkeit, Daten textbasiert, beispielsweise in Form von JSON-Dokumenten, zu speichern.

Daher wäre es denkbar, mit GSON serialisierte Objektdaten direkt in einer solchen Datenbank abzulegen, wobei die Objekt-Metadaten am Kopf des jeweiligen Dokuments ergänzt werden könnten. Da die Schema-Informationen der Objektdaten auch bei der relationalen Darstellung bereits mit abgelegt werden (siehe Abschnitt 6.4), ergibt sich nur eine geringfügige Erhöhung des benötigten Speicherplatzes (durch die Schema-Informationen der sonstigen Metadaten).

Document Stores erlauben Abfragen auf Basis aller im Dokument vorhandenen Attribute, dementsprechend könnten Objekte hierbei, analog zum SDC, nach bestimmten Werten gefiltert werden. Dieser Vorteil überwiegt den geringfügigen Speicherzuwachs bei weitem.

Im Gegensatz zu relationalen Datenbanken erzwingen Document Stores jedoch kein festes Schema, demnach besteht kein Problem darin, verschiedenste Arten von Objektstrukturen abzulegen. Eine Anfrage basierend auf bestimmten Attributen berücksichtigt dabei nur Datensätze, die das jeweilige Feld enthalten; dabei wären sogar Anfragen über mehrere Objektklassen hinweg möglich.

Dieser Ansatz ist jedoch nicht mit der in Abschnitt 3.3 beschriebenen feingranularen Verschlüsselung kompatibel, da alle Werte innerhalb des Objekts potentiell vom Datenbanksystem gelesen werden müssen. Demnach müsste hier der vom SDC implementierte Ansatz der vollständigen Datenbankverschlüsselung genutzt werden.

Literaturverzeichnis

[Aro12] M. Arora. *How secure is AES against brute force attacks?* 7. Mai 2012. URL: http://www.eetimes.com/document.asp?doc_id=1279619 (zitiert auf S. 49).

[Azi13] S. Aziz. *Secure your content provider with SQLCipher.* 6. Nov. 2013. URL: http://sohailaziz05.blogspot.ch/2013/11/secure-your-content-provider-with.html (zitiert auf S. 19).

[BCG13] K. Benton, L. J. Camp, V. Garg. „Studying the effectiveness of android application permissions requests“. In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on.* IEEE. 2013, S. 291–296 (zitiert auf S. 12, 20).

[BJL+13] R. Balebako, J. Jung, W. Lu, L. F. Cranor, C. Nguyen. „Little brothers watching you: Raising awareness of data leaks on smartphones“. In: *Proceedings of the Ninth Symposium on Usable Privacy and Security.* ACM. 2013, S. 12 (zitiert auf S. 12).

[BKOS10] D. Barrera, H. G. Kayacik, P. C. van Oorschot, A. Somayaji. „A methodology for empirical analysis of permission-based security models and its application to android“. In: *Proceedings of the 17th ACM conference on Computer and communications security.* ACM. 2010, S. 73–84 (zitiert auf S. 15).

[Bre00] E. A. Brewer. „Towards robust distributed systems“. In: *PODC.* Bd. 7. 2000 (zitiert auf S. 47).

[Bre13] P. Breault. *Parcelable vs Serializable.* 18. Apr. 2013. URL: http://www.developerphil.com/parcelable-vs-serializable/ (zitiert auf S. 18).

[CBJP11] H. Choe, J. Baek, H. Jeong, S. Park. „MetaService: an object transfer platform between Android applications“. In: *Proceedings of the 2011 ACM Symposium on Research in Applied Computation.* ACM. 2011, S. 56–60 (zitiert auf S. 22).

[CFGW11] E. Chin, A. P. Felt, K. Greenwood, D. Wagner. „Analyzing inter-application communication in Android“. In: *Proceedings of the 9th international conference on Mobile systems, applications, and services.* ACM. 2011, S. 239–252 (zitiert auf S. 18).

[Cod70] E. F. Codd. „A relational model of data for large shared data banks“. In: *Communications of the ACM* 13.6 (1970), S. 377–387 (zitiert auf S. 46).

[CRP14] T. Cooijmans, J. de Ruiter, E. Poll. „Analysis of secure key storage solutions on Android“. In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices.* ACM. 2014, S. 11–20 (zitiert auf S. 37, 50).

[CS11] S. Chatterjee, P. Sarkar. *Identity-based encryption*. Springer Science & Business Media, 2011 (zitiert auf S. 24).

[DDSW10] L. Davi, A. Dmitrienko, A.-R. Sadeghi, M. Winandy. „Privilege escalation attacks on android“. In: *International Conference on Information Security*. Springer. 2010, S. 346–360 (zitiert auf S. 12, 20).

[dev17a] developer.android.com. *Android Debug Bridge*. 2017. URL: https://developer.android.com/studio/command-line/adb.html (zitiert auf S. 36).

[dev17b] developer.android.com. *Android Interface Definition Language (AIDL)*. 2017. URL: https://developer.android.com/guide/components/aidl.html (zitiert auf S. 40).

[dev17c] developer.android.com. *Android Keystore System*. 2017. URL: https://developer.android.com/training/articles/keystore.html (zitiert auf S. 49).

[dev17d] developer.android.com. *Content Provider Basics*. 2017. URL: https://developer.android.com/guide/topics/providers/content-provider-basics.html (zitiert auf S. 18, 19).

[dev17e] developer.android.com. *Copy and Paste*. 2017. URL: https://developer.android.com/guide/topics/text/copy-paste.html (zitiert auf S. 20).

[dev17f] developer.android.com. *Creting a Content Provider*. 2017. URL: https://developer.android.com/guide/topics/providers/content-provider-creating.html (zitiert auf S. 19).

[dev17g] developer.android.com. *Intent*. 2017. URL: https://developer.android.com/reference/android/content/Intent.html (zitiert auf S. 17).

[dev17h] developer.android.com. *Intents and Intent Filters*. 2017. URL: https://developer.android.com/guide/components/intents-filters.html (zitiert auf S. 16).

[dev17i] developer.android.com. *javax.crypto*. 2017. URL: https://developer.android.com/reference/javax/crypto/package-summary.html (zitiert auf S. 48).

[dev17j] developer.android.com. *Parcel*. 2017. URL: https://developer.android.com/reference/android/os/Parcel.html (zitiert auf S. 17, 42).

[dev17k] developer.android.com. *Parcelable*. 2017. URL: https://developer.android.com/reference/android/os/Parcelable.html (zitiert auf S. 42).

[dev17l] developer.android.com. *<permission>*. 2017. URL: https://developer.android.com/guide/topics/manifest/permission-element.html (zitiert auf S. 20).

[dev17m] developer.android.com. *Requesting Permissions at Run Time*. 2017. URL: https://developer.android.com/training/permissions/requesting.html (zitiert auf S. 20).

[dev17n] developer.android.com. *Saving Files*. 2017. URL: https://developer.android.com/training/basics/data-storage/files.html (zitiert auf S. 15).

[dev17o] developer.android.com. *Saving Key-Value Sets*. 2017. URL: https://developer.android.com/training/basics/data-storage/shared-preferences.html (zitiert auf S. 45).

[dev17p] developer.android.com. *Services.* 2017. URL: https://developer.android.com/guide/components/services.html (zitiert auf S. 39).

[dre17] dre.vanderbilt.edu. *Designing a Remote Interface Using AIDL.* 2017. URL: http://www.dre.vanderbilt.edu/~schmidt/android/android-4.0/out/target/common/docs/doc-comment-check/guide/developing/tools/aidl.html (zitiert auf S. 60).

[EGH+14] W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014), S. 5 (zitiert auf S. 11, 26).

[fut17a] futuremark.com. *3DMark.* 2017. URL: https://www.futuremark.com/benchmarks/3dmark/all (zitiert auf S. 70).

[fut17b] futuremark.com. *Asus ZenFone Max Review.* 2017. URL: https://www.futuremark.com/hardware/mobile/Asus+ZenFone+Max/review (zitiert auf S. 70).

[fut17c] futuremark.com. *LG G4 Review.* 2017. URL: https://www.futuremark.com/hardware/mobile/LG+G4/review (zitiert auf S. 70).

[fut17d] futuremark.com. *Motorola Moto E 4G LTE (2nd Gen) Review.* 2017. URL: https://www.futuremark.com/hardware/mobile/Motorola+Moto+E+4G+LTE+%282nd+Gen%29/review (zitiert auf S. 70).

[Gar14] H. Garg. *7 differences between Serializable and Externalizable interface in Java.* 19. Dez. 2014. URL: http://www.codingeek.com/java/io/differences-serializable-externalizable-interface-java-tutorial/ (zitiert auf S. 41).

[GJG+11] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, H. M. Levy. „Keypad: An auditing file system for theft-prone devices“. In: *Proceedings of the sixth conference on Computer systems.* ACM. 2011, S. 1–16 (zitiert auf S. 51).

[Goo17] Google. *Gson User Guide.* 2017. URL: https://github.com/google/gson/blob/master/UserGuide.md (zitiert auf S. 43).

[HSG11] N. Husted, H. Saidi, A. Gehani. „Smartphone security limitations: conflicting traditions“. In: *Proceedings of the 2011 Workshop on Governance of Technology, Information, and Policies.* ACM. 2011, S. 5–12 (zitiert auf S. 38).

[Kha16] M. H. Khan. *Securely store app data in Android - Android KeyStore to the rescue!* 16. Dez. 2016. URL: https://devliving.online/securely-store-preference-data-in-android/ (zitiert auf S. 62).

[Kru16] Krumelur. *Launching an Android Activity of another APK.* 6. Okt. 2016. URL: https://krumelur.me/2015/10/06/launching-an-android-activity-of-another-apk/ (zitiert auf S. 17).

[McC91] J. McCumber. „Information systems security: A comprehensive model“. In: *Proceedings of the 14th National Computer Security Conference.* National Institute of Standards und Technology. 1991 (zitiert auf S. 13).

[New10] T. Neward. *5 things you didn't know about ... Java Object Serialization.* 6. Apr. 2010. URL: https://www.ibm.com/developerworks/library/j-5things1/ (zitiert auf S. 16).

[NPP13] A. Nayak, A. Poriya, D. Poojary. „Type of NOSQL databases and its comparison with relational databases". In: *International Journal of Applied Information Systems* 5.4 (2013), S. 16–19 (zitiert auf S. 47).

[OBM10] M. Ongtang, K. Butler, P. McDaniel. „Porscha: Policy oriented secure content handling in Android". In: *Proceedings of the 26th Annual Computer Security Applications Conference.* ACM. 2010, S. 221–230 (zitiert auf S. 24).

[Ora15] Oracle. *Serializable Objects.* 2015. URL: https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html (zitiert auf S. 15, 41).

[Ora17] Oracle. *Interface Externalizable.* 2017. URL: https://docs.oracle.com/javase/7/docs/api/java/io/Externalizable.html (zitiert auf S. 41).

[Pok13] J. Pokorny. „NoSQL databases: a step to database scalability in web environment". In: *International Journal of Web Information Systems* 9.1 (2013), S. 69–82 (zitiert auf S. 47).

[Reh12] A. Rehman. *How To Bypass/Disable Pattern Unlock On Android via ADB Commands.* 18. Aug. 2012. URL: http://www.addictivetips.com/android/how-to-bypass-disable-pattern-unlock-on-android-via-adb-commands/ (zitiert auf S. 37).

[RSS15] E. Rahm, G. Saake, K.-U. Sattler. „Konsistenz in Cloud-Datenbanken". In: *Verteiltes und Paralleles Datenmanagement.* Springer, 2015, S. 353–369 (zitiert auf S. 47).

[SFK+10] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, C. Glezer. „Google android: A comprehensive security assessment". In: *IEEE Security & Privacy* 8.2 (2010), S. 35–44 (zitiert auf S. 37).

[Shi09] K. Shilton. „Four billion little brothers?: Privacy, mobile phones, and ubiquitous data collection". In: *Communications of the ACM* 52.11 (2009), S. 48–53 (zitiert auf S. 11).

[sim13] simple.sourceforge.net. *Simple XML Serialization Tutorial.* 2013. URL: http://simple.sourceforge.net/download/stream/doc/tutorial/tutorial.php (zitiert auf S. 42).

[Sim79] G. J. Simmons. „Symmetric and asymmetric encryption". In: *ACM Computing Surveys (CSUR)* 11.4 (1979), S. 305–330 (zitiert auf S. 48).

[SKS13] D. Schreckling, J. Köstler, M. Schaff. „Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android". In: *information security technical report* 17.3 (2013), S. 71–80 (zitiert auf S. 26).

[SM13] C. Stach, B. Mitschang. „Privacy management for mobile platforms–a review of concepts and approaches". In: *Mobile Data Management (MDM), 2013 IEEE 14th International Conference on.* Bd. 1. IEEE. 2013, S. 305–313 (zitiert auf S. 27, 55).

[SM14] C. Stach, B. Mitschang. „Design and implementation of the privacy management platform". In: *Mobile Data Management (MDM), 2014 IEEE 15th International Conference on.* Bd. 1. IEEE. 2014, S. 69–72 (zitiert auf S. 27, 55, 56).

[SM15] C. Stach, B. Mitschang. „Der Secure Data Container (SDC)“. In: *Datenbank-Spektrum* 15.2 (2015), S. 109–118 (zitiert auf S. 27, 28).

[SM16] C. Stach, B. Mitschang. „The Secure Data Container: An Approach to Harmonize Data Sharing with Information Security“. In: *Mobile Data Management (MDM), 2016 17th IEEE International Conference on*. Bd. 1. IEEE. 2016, S. 292–297 (zitiert auf S. 27).

[SMA+07] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, P. Helland. „The end of an architectural era:(it's time for a complete rewrite)“. In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, S. 1150–1160 (zitiert auf S. 46).

[Son14] M. Sonar. *What are the advantage and disadvantage of shared preference in Android?* 2014. URL: https://www.quora.com/What-are-the-advantage-and-disadvantage-of-shared-preference-in-Android (zitiert auf S. 45).

[sql17] sqlite.org. *PRAGMA Statements*. 2017. URL: http://www.sqlite.org/pragma.html#pragma_secure_delete (zitiert auf S. 51, 71).

[Sta17a] Statista. *Anzahl der in Gebrauch befindlichen Smartphones weltweit nach Betriebssystem im März 2017*. 2017. URL: https://de.statista.com/statistik/daten/studie/246004/umfrage/weltweiter-bestand-an-smartphones-nach-betriebssystem/ (zitiert auf S. 39).

[Sta17b] Statista. *Anzahl der Smartphone-Nutzer in Deutschland in den Jahren 2009 bis 2016*. 2017. URL: https://de.statista.com/statistik/daten/studie/198959/umfrage/anzahl-der-smartphonenutzer-in-deutschland-seit-2010/ (zitiert auf S. 11).

[SY06] S. Subramanya, B. K. Yi. „Digital rights management“. In: *IEEE Potentials* 25.2 (2006), S. 31–34 (zitiert auf S. 24).

[Tah16] S. Tahiri. *Mobile Forensics*. 2016. URL: https://www.packtpub.com/books/content/mobile-forensics (zitiert auf S. 50).

[Tam08] A.-K. A. Tamimi. *Performance Analysis of Data Encryption Algorithms*. 2008. URL: http://www.cs.wustl.edu/~jain/cse567-06/encryption_perf.htm (zitiert auf S. 48).

[tre17] trends.google.com. *Search Trends: XML & JSON*. 2017. URL: https://trends.google.com/trends/explore?q=xml,json (zitiert auf S. 43).

[VVC11] T. Vidas, D. Votipka, N. Christin. „All Your Droid Are Belong to Us: A Survey of Current Android Attacks.“ In: *WOOT*. 2011, S. 81–90 (zitiert auf S. 36).

[VZC11] T. Vidas, C. Zhang, N. Christin. „Toward a general collection methodology for Android devices“. In: *digital investigation* 8 (2011), S14–S24 (zitiert auf S. 37).

[XSA12] R. Xu, H. Saidi, R. J. Anderson. „Aurasium: practical policy enforcement for android applications.“ In: *USENIX Security Symposium*. Bd. 2012. 2012 (zitiert auf S. 11, 38).

Alle URLs wurden zuletzt am 09. 07. 2017 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