

Hochschule Bochum
Fachbereich Elektrotechnik und Informatik
WS 2015/16
Prof. Dr. rer. nat. Rainer Lütticke

Bachelorarbeit

Entwicklung eines Frameworks zur Verschlüsselung von Daten in Android mit automatischem Schlüsselmanage- ment

von Daniel Sahm

daniel.sahm@hs-bochum.de
Studiengang B. Sc. Informatik
Matrikelnummer 01220 2140

Inhaltsverzeichnis

1. Einleitung	3
1.1. Hintergrund	3
1.2. Ziele der Arbeit	3
1.3. Aufbau der Arbeit.....	5
2. Motivation	6
2.1. Angriffe auf Android.....	6
2.2. Unsaubere Implementierung von Kryptographie in Android	6
2.3. Gesetzliche Vorgaben.....	7
3. Kryptographische Grundlagen	9
3.1. Ziele von Kryptographie	9
3.2. Kryptographische Grundbegriffe	9
3.3. Angriffe auf Verschlüsselungsverfahren	14
3.3.1. Generische Betrachtung eines Kryptosystems	15
3.3.2. Konkrete Angriffe auf Verfahren.....	18
3.3.3. Modellierung von Gefahren	19
4. Realisierung	22
4.1. Überblick Framework.....	22
4.2. API	28
4.3. Verschlüsselung	41
4.3.1. Private encrypt-Methode	42
4.4. Entschlüsselung	57
4.4.1. Private decrypt-Methode	57
4.5. Speichern, Laden und Löschen.....	60
4.5.1. Verschlüsseln und Speichern	60

4.5.2. Laden und Entschlüsseln	62
4.5.3. Gezieltes Löschen.....	67
4.5.4. Alle Daten löschen	67
4.6. Testen	67
4.6.1. Unit-Tests	68
4.6.2. Testen mit einer Beispiel-App.....	73
4.6.3. Testen mit der eap-App Stadtwerkzeug.....	75
4.7. Produktive Nutzung in eap-Apps	79
4.7.1. Übergang zu SecureAndroid	79
4.7.2. Asynchrones Ver- und Entschlüsseln	81
4.7.3. Größenänderung der Apps	84
5. Vergleich mit Alternativen	85
5.1. Facebook-Conceal	85
5.2. SQL-Cipher	85
5.3. Klassenbibliotheken	86
5.4. Fazit des Vergleichs	86
6. Zusammenfassung und Fazit.....	87
6.1. Aktueller Projektstatus	88
6.2. Ausblick.....	90
Literaturverzeichnis.....	91
Anhang	95

1. Einleitung

1.1. Hintergrund

Im Jahr 2014 wurden weltweit 1.06 Milliarden Smartphones und 229.6 Millionen Tablets verkauft [3] [21]. Prognosen gehen davon aus, dass die Verkaufszahlen in den kommenden Jahren weiter steigen werden [40]. Etwa 75 % der verkauften Geräte werden mit dem Betriebssystem Android ausgeliefert (s. Praxisphasenbericht Kap. 4). Aufgrund des hohen Verbreitungsgrades ist das von Google entwickelte Betriebssystem ein attraktives Ziel für Hacker und Malware-Entwickler. Gleichzeitig existiert keine Klassenbibliothek, um Daten in Android einfach und sicher zu verschlüsseln (s. Kap. 5). Auch das manuelle Anwenden von kryptographischen Algorithmen birgt Gefahren, welche die Verschlüsselung unsicher machen können (s. Kap. 2). Aus diesen Gründen wird in dieser Arbeit ein Framework zur Datenverschlüsselung in Android entwickelt. Das Framework verwaltet kryptographische Schlüssel automatisch und arbeitet nach dem Prinzip des *Intermediate-Keys* (s. Kap. 1.2). Das Framework wird ab hier mit *SecureAndroid* bezeichnet. In den von der *energy-app-provider GmbH (eap)*¹ entwickelten Apps soll das Framework dazu eingesetzt werden, personenbezogene Daten vor unbefugtem Zugriff zu schützen. Zusätzlich wird *SecureAndroid* als Open-Source-Bibliothek veröffentlicht, um anderen Entwicklern die Nutzung zu ermöglichen.

1.2. Ziele der Arbeit

Mit *SecureAndroid* soll eine einfach zu benutzende und gut dokumentierte Klassenbibliothek entstehen, die es Android-Entwicklern ermöglicht, ohne das Wissen über kryptographische Implementierungsdetails sichere Datenverschlüsselung in Android durchführen zu können. Die Verschlüsselungstechniken sollen dabei die Anforderungen des Bundesdatenschutzgesetzes erfüllen [7].

Abb. 1 zeigt eine grobe Strukturübersicht der Anforderungen an die interne Arbeitsweise des Frameworks. Wichtig hierbei ist das Prinzip des Intermediate-Keys (dt. Zwischenschlüssel). Das von dem Nutzer oder App-Entwickler bereit gestellte Passwort wird zur Generierung eines Master-Keys verwendet. Dieser Key wird nicht auf dem Gerät gespeichert, sondern bei jeder Passworteingabe aus dem eingegebenen Passwort generiert. Mit diesem Master-Key wird ein Intermediate-Key verschlüsselt und abgespeichert. Der Intermediate-Key wird für die Ver- und Entschlüsselung der eigentlichen Nutzdaten verwendet und bei jedem Ver- und Entschlüsselungsvorgang mit dem Master-Key entschlüsselt.

¹ <http://www.energy-app-provider.com>

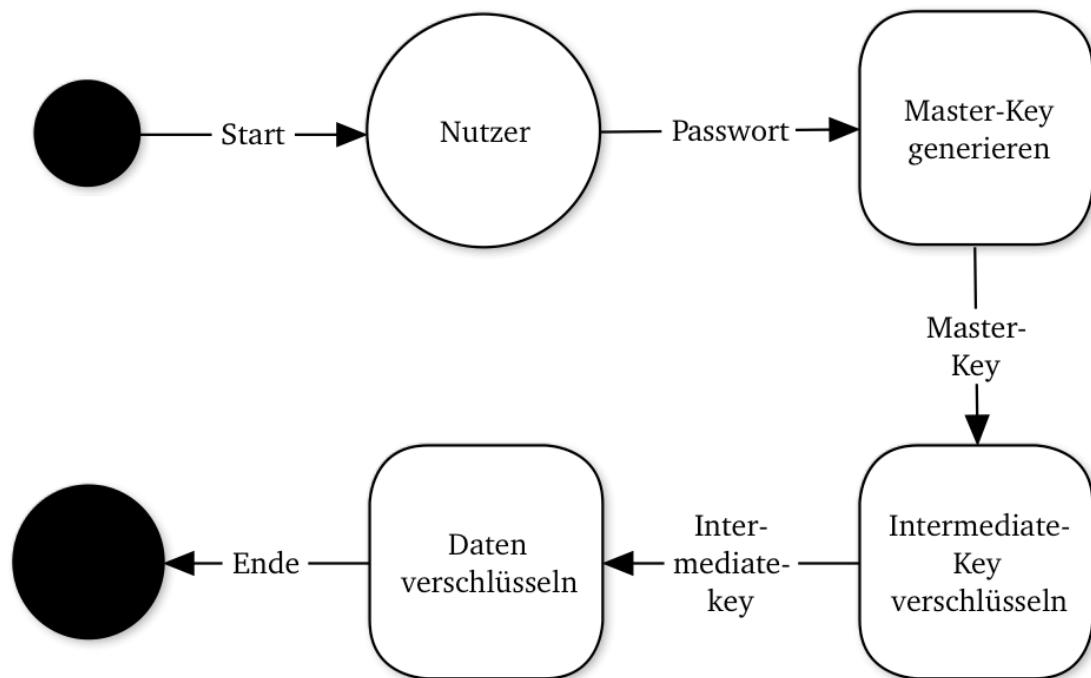


Abbildung 1: Strukturübersicht der Anforderungen an die Schlüsselverwaltung

Diese Vorgehensweise bietet folgende Vorteile:

A. Der Master-Key wird nicht auf dem Gerät gespeichert

Ein nicht auf dem Gerät gespeicherter Schlüssel kann nicht entwendet oder auf andere Weise kompromittiert werden.

B. Einfaches Löschen der verschlüsselten Daten

Wenn die verschlüsselten Daten gelöscht werden sollen - z. B. wenn eine das Framework benutzende App eine Kompromittierung des Gerätes entdeckt -, müssen nur die Intermediate-Keys gelöscht werden. Da die Daten nur mit diesen Keys entschlüsselt werden können, sind sie unbrauchbar gemacht.

C. Einfaches Ändern des Passworts

Wenn das Passwort zum Verschlüsseln des Master-Keys verändert werden soll, müssen nicht die verschlüsselten Daten, sondern nur die Intermediate-Keys mit dem alten Passwort entschlüsselt und mit dem neuen Passwort wieder verschlüsselt werden.

Des Weiteren soll das Framework die Möglichkeit bieten, Daten nach dem Verschlüsseln direkt abzuspeichern. Dies soll für den Aufrüfer der Methode transparent geschehen. Das Framework dient also neben dem Verbergen von kryptographischen Implementierungsdetails auch der Komplexitätsreduktion von I/O-Operationen. Schlussendlich soll das Framework leicht in ein Android-Projekt eingebunden werden können und die Größe einer App nicht aufblähen.

1.3. Aufbau der Arbeit

Diese Arbeit erläutert in Kapitel 2 zunächst die Motivation und Notwendigkeit für das Schreiben einer Verschlüsselungsbibliothek für Android. In Kapitel 3 werden die grundlegenden kryptographischen Begriffe und Verfahren, die zum Verständnis von SecureAndroid notwendig sind, erläutert. Das vierte Kapitel erklärt die Realisierung des Frameworks anhand von Code-Beispielen, des Zusammenhangs der einzelnen Klassen sowie detaillierten Erläuterungen der angewandten kryptographischen Mechanismen. Darauf folgend werden im fünften Kapitel vergleichbare Klassenbibliotheken vorgestellt und mit SecureAndroid verglichen. Das letzte Kapitel zeigt den aktuellen Projektstatus auf und beschäftigt sich mit dem Ausblick in die Zukunft.

2. Motivation

In diesem Kapitel wird die grundsätzliche Notwendigkeit für das Schreiben einer Verschlüsselungsbibliothek zum Schutz von sensiblen Daten auf Android-Geräten erläutert. Der konkrete Vergleich von SecureAndroid mit Alternativen findet in Kapitel 5 statt.

2.1. Angriffe auf Android

Aufgrund des hohen Verbreitungsgrades (s. Kap. 1.1) ist die Android-Plattform ein immer attraktiver werdendes Ziel für Malware-Entwickler und Hacker. Laut einem Bericht des Security-Unternehmens Sophos aus dem Jahr 2014 existierten zu diesem Zeitpunkt 650.000 Android-Schadprogramme [37]. Im Gegensatz zu Apple erlaubt Google den Nutzern von Android, Apps außerhalb des hauseigenen App-Stores *Google-Play*² zu installieren. Diese Offenheit führt auf der einen Seite zu mehr Freiraum für Android-Nutzer, auf der anderen Seite aber auch zu der Gefahr, Apps mit Schadcode zu installieren [15]. Überdies werden in Android immer wieder gravierende Sicherheitslücken entdeckt. Exemplarisch dafür ist die *Stagefright*-Lücke, die im Juli 2015 veröffentlicht wurde [38]. Diese Lücke betraf fast 95 % aller Android-Geräte und ließ sich bereits durch das Senden einer MMS an ein Opfer ausnutzen. Laut einer Studie existieren in über der Hälfte aller Android-Geräte nicht behobene Sicherheitslücken [26]. Die Fragmentierung des Android-Markts (s. Praxisphasenbericht Kap. 4) führt dazu, dass Hersteller ihre eigenen Android-Versionen häufig nicht mit Sicherheitsupdates versorgen.

2.2. Unsaubere Implementierung von Kryptographie in Android

Das IT-Security Unternehmen FireEye hat im November 2014 alle 9339 kostenlosen Apps aus dem Google-Playstore mit über einer Million Downloads auf die korrekte Implementierung von kryptographischen Algorithmen überprüft [29]. Von diesen Apps nutzen 8261 kryptographische Funktionalitäten, welche durch die Android-API zur Verfügung gestellt werden. Von den überprüften Apps machen 5147 grobe Implementierungsfehler. Dadurch ist die Verschlüsselung von Daten sowohl während des Transports durch Netzwerke als auch nach der Speicherung auf einem Gerät angreifbar. Die zu dieser Situation führende Problematik ist mehrschichtig: Auf der einen Seite betreibt die Android-API den AES-Algorithmus standardmäßig in dem unsicheren ECB-Modus (s. Praxisphasenbericht Kap. 2). Andererseits fehlt vielen Entwicklern die Kenntnis über das korrekte Anwenden von Verschlüsselungsalgorithmen. Angewandte Kryptographie ist ein Spezialgebiet. Ein App-Entwickler, der Business-Logik entwickelt, hat häufig nicht genügend Zeit, sich in diese Details einzuarbeiten [30]. Wenn die Anforderung an

² <http://play.google.com>

Verschlüsselung dennoch gegeben ist, suchen viele Entwickler in Internet-Foren wie *Stackoverflow* nach Beispiel-Code. Die Antworten auf Fragen bezüglich der Kryptographie ist in diesen Foren allerdings häufig von zweifelhafter Qualität, siehe [19]. Selbst in einem Android-Fachbuch zur App-Entwicklung wird der AES-Algorithmus im unsicheren ECB-Modus initialisiert [8].

Im Rahmen einer weiteren Studie [18] über fehlerhafte Implementierung von Kryptographie in Android-Apps aus dem Jahr 2013 wurden 11.748 Android-Apps mit kryptographischen Funktionen untersucht. Von den 11.748 Apps machen 88 % mindestens einen der folgenden grundlegenden Fehler:

1. Der AES-Algorithmus wird im ECB-Modus betrieben (s. oben und Praxisphasenbericht Kap. 2).
2. Initialisierungsvektoren (IVs, s. Praxisphasenbericht Kap. 2.3) werden nicht für jeden Verschlüsselungsvorgang zufällig generiert, sondern als Konstante im Code gespeichert.
3. Kryptographische Schlüssel werden unverschlüsselt als Konstanten im Code gespeichert.
4. Salt-Werte (s. Kap. 3.2) werden nicht zufällig generiert, sondern als Konstanten im Code gespeichert.
5. PBKD-Funktionen (s. Kap. 3.2) werden mit weniger als 1000 Iterationen aufgerufen.
6. Für Pseudo-Zufallszahlengeneratoren (s. Kap. 3.2) werden statische Seed-Werte benutzt.

Jeder dieser Fehler führt bereits einzeln zu unsicherer Verschlüsselung. Zusammenfassend lässt sich festhalten, dass der Bedarf an einem Framework, das Entwicklern einfach zu benutzende und sichere Verschlüsselungsfunktionen in Android bietet, sehr groß ist.

2.3. Gesetzliche Vorgaben

Anbieter von Smartphone-Apps mit Sitz in Deutschland müssen sich wie alle anderen datenverarbeitenden Stellen gemäß § 1 Bundesdatenschutzgesetz (BDSG) an die deutschen Datenschutzgesetze halten. Beim Datenschutz geht es um *personenbezogene Daten*. Dies sind Daten, die entweder direkt oder mit Hilfe weiterer Daten auf eine Person zurückgeführt werden können. Beispiele dafür sind:

- ◆ IP-Adresse
- ◆ Standortdaten
- ◆ Stimmaufnahmen
- ◆ Biometrische Daten
- ◆ Eindeutige Gerätekennungen
- ◆ Adressdaten

Das BDSG definiert "*Technische und organisatorische Maßnahmen*", um personenbezogene Daten zu schützen [7]. Dazu gehört die Vorgabe, dass lokal gespeicherte Daten nach aktuellem Stand der Technik mit kryptographischen Verfahren abgesichert werden müssen [25]. Der Verstoß gegen die Richtlinien des BDSG stellt eine Ordnungswidrigkeit und unter Umständen sogar einen Straftatbestand dar. Ein Bußgeld in Höhe von bis zu 300.000 EUR kann die Folge sein. Neben dem Schutz vor Malware- und Hacker-Angriffen ist eine kryptographische Absicherung von sensiblen Daten somit auch notwendig, damit ein App-Anbieter nicht mit der deutschen Gesetzgebung in einen Konflikt gerät.

3. Kryptographische Grundlagen

In diesem Kapitel werden kryptographische Grundlagen gelegt und wichtige Begriffe, die zum Verständnis dieser Arbeit notwendig sind, erläutert. In Kapitel 3.1 werden die Ziele von Kryptographie erklärt. In Kapitel 3.2 werden die für diese Arbeit relevanten grundlegenden Begriffe und Konzepte der Kryptographie vorgestellt und Kapitel 3.3 behandelt Angriffe auf Verschlüsselungsverfahren.

3.1. Ziele von Kryptographie

Kryptographie wird angewendet, um eines oder mehrere der folgenden Ziele zu erreichen [16]:

- ◆ **Vertraulichkeit**

Vertraulichkeit von Daten bedeutet, dass nur die Person, für welche die Daten bestimmt sind, diese auch lesen kann.

- ◆ **Integrität**

Integrität von Daten bedeutet, dass die Daten genau in der Form vorliegen, die sie bei Speicherung bzw. nach dem Erstellen durch den Absender hatten.

- ◆ **Authentizität**

Bei der Authentizität von Daten geht es darum, mit Sicherheit überprüfen zu können, dass die empfangenen Daten auch tatsächlich von dem angegebenen Absender bzw. Ersteller stammen.

- ◆ **Nicht-Abstreitbarkeit**

Wenn bei einem Nachrichtenaustausch Nicht-Abstreitbarkeit gegeben sein soll, darf es einem Urheber einer Nachricht nicht möglich sein, die Urheberschaft zu leugnen.

In SecureAndroid werden Vertraulichkeit, Integrität und Authentizität von Daten gefordert. Die Verschlüsselung mit AES im CBC-Modus sorgt für die Vertraulichkeit der Daten und das Anwenden eines geeigneten Verfahrens zur Erstellung eines Message Authentication Codes (MAC) (s. Kap. 3.2) stellt deren Integrität und Authentizität sicher. Weil es um das verschlüsselte Abspeichern von Daten und nicht um einen Nachrichtenaustausch geht, wird Nicht-Abstreitbarkeit nicht benötigt.

3.2. Kryptographische Grundbegriffe

- ◆ **AES-Verschlüsselung und Betriebsmodi**

Für das Grundlagenverständnis der AES-Verschlüsselung und der Auswahl eines sicheren Betriebsmodus verweise ich auf den Praxisphasenbericht, Kap. 2.

◆ Pseudo-Random-Number-Generator

Ein Pseudo-Random-Number-Generator (PRNG) ist ein Generator von Zahlen, welche Zufallszahlen sehr ähnlich sind. Die Zahlen sind nicht wirklich zufällig, da die Generator-Funktion als Seed-Werte nur eine begrenzte Menge von Variablen aufnehmen kann. Unter Seed-Wert ist der Input für die Generator-Funktion zu verstehen. Kryptographisch sichere PRNGs (CSPRNGs) sind ein wichtiger Grundbaustein für viele kryptographische Algorithmen, z. B. für das Berechnen eines IVs. Ein CSPRNG liefert Zufallszahlen so, dass ein Angreifer diese von echten Zufallszahlen nicht unterscheiden kann [17]. Diese Eigenschaft eines CSPRNGs wird über folgende Anforderungen definiert:

1. Bestehen des *Next-Bit-Tests* [17]

Sind einem Angreifer n Zufallszahlen bekannt, so ist es ihm nicht möglich, mit polynomialem Aufwand die $n+1$ -te Zahl mit einer Erfolgswahrscheinlichkeit größer 50 % zu berechnen.

2. Bestehen des *State-Compromise-Extension-Angriffs* [17]

Wenn ein Angreifer den inneren Zustand eines CSPRNG kennt, darf er anhand dieser Information keine Aussagen über den Output des CSPRNG vor und nach dem bekannten Zustand ableiten können.

In SecureAndroid werden kryptographisch sichere Zufallszahlen mit der Methode `nextBytes` aus der Klasse `SecureRandom` der Java-Standard-API generiert. Als Seed-Generator wird die von Android bereit gestellte Entropie-Quelle `/dev/random` bzw. `/dev/urandom` genutzt (s. Kap. 4.1).

◆ Hash-Funktion

Hash-Funktionen sind mathematische Kompressionsfunktionen und bilden einen Wert beliebiger Länge auf einen Wert fester Länge ab [16]. Eine Hash-Funktion, die in der Kryptographie angewendet wird, sollte dabei für zwei Eingabewörter, welche sich in nur einem Bit unterscheiden, zwei völlig verschiedene Ausgabewerte liefern. Weiter muss es praktisch unmöglich sein, vom Ausgabewert auf den Eingabewert zu schließen. Hash-Funktionen sind folglich nicht injektiv. Kryptographisch sichere Hash-Funktionen müssen je nach Anwendung eine oder mehrere der folgenden Bedingungen erfüllen [13]:

1. Einweg-Eigenschaft

Es muss praktisch unmöglich sein, zu einem gegebenen Hash-Wert y einen Eingabewert x mit $\text{hash}(x) = y$ zu finden. Dies nennt man auch *preimage resistance*.

2. Schwache Kollisionsresistenz

Es muss praktisch unmöglich sein, zu einem gegebenen Eingabewert x ein y mit $\text{hash}(x) = \text{hash}(y)$ zu finden. Dies nennt man auch *second-preimage resistance*.

3. Starke Kollisionsresistenz

Es muss praktisch unmöglich sein, zwei verschiedene Eingabewerte x und y mit $\text{hash}(x) = \text{hash}(y)$ zu finden. Dies bedeutet nicht, dass keine Kollisionen existieren dürfen, sondern dass es sehr schwierig sein muss, diese zu finden.

Eine Hash-Funktion, die alle drei Bedingungen erfüllt, gilt als *kryptographisch stark* [13]. In SecureAndroid wird zu folgenden Zwecken eine Hash-Funktion benötigt:

1. Sicheres Ablegen eines Passworts
2. Grundlage für ein geeignetes MAC-Verfahren
3. Grundlage für eine Password-Based-Key-Derivation-Funktion (s. unten)

Zur Auswahl einer geeigneten Hash-Funktion für SecureAndroid wird auf den Unterpunkt *Empfehlungen des BSI für Schlüssellängen und Algorithmen* verwiesen (s. unten).

◆ Salt

Ein Salt-Wert ist wie ein IV (s. Praxisphasenbericht Kap. 2.3) eine Folge von Zufallsbits. Ein Salt wird für die sichere Speicherung von Passwörtern verwendet. Passwörter sollten niemals im Klartext gespeichert werden. Stattdessen wird das Passwort mit dem Salt-Wert verknüpft und dann mittels einer Hash-Funktion auf einen Wert fester Länge abgebildet. Dieser Hash-Wert wird zusammen mit dem Salt gespeichert. Will ein Nutzer sich nun mit seinem Passwort bei einem Webdienst oder in einer Anwendung anmelden, wird das Passwort wieder mit dem Salt zusammengefügt und der Hash-Funktion übergeben. Ist das Ergebnis der erneuten Berechnung mit dem gespeicherten Hash-Wert identisch, hat der User das richtige Passwort eingegeben.

Durch diese Vorgehensweise kann ein Angreifer, der gespeicherte Hash-Werte erbeutet, die Passwörter nicht mittels einer *Rainbow-Table* (s. Kap. 3.3.2) extrahieren. Der Hash-Wert eines Passworts ist durch die Verknüpfung des Passworts mit dem Salt völlig anders als der Hash-Wert des Passworts ohne Salt. In SecureAndroid werden Passwörter aus diesem Grund vor der Speicherung mit einem Salt verknüpft. Zusätzlich wird ein Salt in SecureAndroid verwendet, um mittels einer Password-Based-Key-Derivation-Funktion aus einem Passwort einen kryptographischen Schlüssel zu erstellen (s. unten). In der PKCS #5-Spezifikation der *Request for Comments (RFCs)* [28] wird empfohlen, Salt-Werte mit einer Länge von mindestens 64 Bit zu verwenden. Weil der Zeitaufwand durch Erstellung eines größeren Salts nicht signifikant erhöht wird, werden in SecureAndroid 512 Bit lange Salts verwendet.

◆ Password-Based-Key-Derivation-Funktion

Wenn Daten mit Hilfe eines Passworts verschlüsselt werden sollen, wird nicht das Passwort als Schlüssel verwendet. Dies wäre sehr schnell erraten, weil viele Nutzer recht einfache Passwörter verwenden [14]. Zusätzlich muss der Schlüssel die richtige Länge für das Verschlüsselungsverfahren aufweisen. Zum Generieren eines Schlüssels aus einem Passwort wird die Technik des *Key-Stretchings* [34] verwendet. Dieses Verfahren ist als *Password-Based-Key-Derivation-Function (PBKDF)* standardisiert. Auch hier wird ein Salt-Wert mit dem Passwort verknüpft. Dann wird das Passwort in einer festgelegten Anzahl an Iterationen immer wieder mittels einer Hash-Funktion gehasht. Der Schlüssel wird also wie folgt berechnet:

$$\text{Key} = \text{Hash}(\text{Hash}(\text{Hash}(\dots(\text{password} + \text{salt}))))$$

Die Anzahl der Iterationen ist der sogenannte *Work-Faktor*. Ein Angreifer, der einen Schlüssel berechnen will, muss ebenfalls den Weg durch die PBKDF gehen. Umso höher der Work-Faktor ist, umso länger muss der Angreifer rechnen. In der PBKDF2-Spezifikation der RFCs [28] wird eine Mindestanzahl von 1000 Iterationen empfohlen. Die Anzahl sollte so gewählt werden, dass die Wartezeit beim Eingeben des Passwortes für den Nutzer noch zumutbar ist, ein Angreifer aber möglichst stark ausgebremst wird [27]. SecureAndroid verwendet eine PBKDF zum Berechnen der Master-Keys aus einem Passwort.

◆ MAC

Ein *MAC* [16] wird verwendet, um die Integrität und Authentizität von Daten sicherzustellen. Zur Generierung eines MACs sind ein geheimer Schlüssel und eine Hash-Funktion notwendig. Der zu schützende Datensatz wird mit dem geheimen Schlüssel verknüpft und dann der Hash-Funktion übergeben. Die Daten werden mit dem berechneten Hash-Wert an den Empfänger gesendet. Dieser ist auch im Besitz des geheimen Schlüssels. So kann er die gleiche Berechnung wie der Versender durchführen und den resultierenden Hash-Wert mit dem empfangenen Wert vergleichen. Sind beide Werte gleich, wurde die Nachricht auf dem Transportweg nicht verändert und die Integrität ist sichergestellt. Weil in die Berechnung ein geheimer Schlüssel einfließt, ist somit auch sichergestellt, dass die Daten von der Person stammen müssen, die im Besitz des geheimen Schlüssels ist. Dies sichert die Authentizität der Daten.

◆ Empfehlungen des BSI für Schlüssellängen und Algorithmen [13]

Das Bundesamt für Sicherheit in der Informationstechnik (BSI) empfiehlt für die Verschlüsselung mit AES die Schlüssellängen 128, 192 oder 256 Bit. SecureAndroid verwendet AES-128. Ein 128 Bit langer Schlüssel kann mit einer Brute-Force-Attacke - also durch das Ausprobieren aller möglichen Schlüssel - auch mit aller auf der Welt verfügbaren Rechenkraft nicht in einem Zeitrahmen gebrochen werden, der kleiner ist als das

Alter des Universums [22]. Auf Smartphones haben Daten meist eine sehr kurze Lebensdauer von wenigen Jahren. Daher würde die Nutzung von AES-192 oder AES-256 in diesem Kontext nur unnötig Rechenzeit verschwenden. AES-256 verbraucht aufgrund der höheren Rundenzahl (14 im Vergleich zu 10) ~30 % mehr Rechenleistung als AES-128.

Für das Erstellen von MACs empfiehlt das BSI die Verfahren CMAC, HMAC (Hashed MAC) [13] und GMAC. Die Verfahren CMAC und GMAC sind auf Android-Geräten nach eigenen Tests im Gegensatz zu HMAC nicht verfügbar. Aus diesem Grund wird in SecureAndroid HMAC verwendet. Tabelle 1 zeigt eine Verfügbarkeitsübersicht von kryptographischen Algorithmen auf den für diese Arbeit getesteten Geräten.

Verfügbarkeit von kryptographischen Algorithmen auf Android-Smartphones		
	MAC	PBKDF
Physische Geräte		
Motorola Moto G	HMACSHA-256	PBKDF2WithHMACSHA-1
Google Nexus 9	HMACSHA-256	PBKDF2WithHMACSHA-1
Samsung Galaxy S5	HMACSHA-256	PBKDF2WithHMACSHA-1
Samsung Galaxy Ace 2	HMACSHA-256	PBKDF2WithHMACSHA-1
Lenovo A5500-H	HMACSHA-256	PBKDF2WithHMACSHA-1
Virtuelle Maschinen		
Google Nexus One	HMACSHA-256	PBKDF2WithHMACSHA-1
Google Nexus 5	HMACSHA-256	PBKDF2WithHMACSHA-1
Google Nexus 5X	HMACSHA-256	PBKDF2WithHMACSHA-1
Google Nexus 6P	HMACSHA-256	PBKDF2WithHMACSHA-1
Google Nexus 9	HMACSHA-256	PBKDF2WithHMACSHA-1
HTC One X	HMACSHA-256	PBKDF2WithHMACSHA-1
LG Optimus L3 II	HMACSHA-256	PBKDF2WithHMACSHA-1
Motorola Droid Razr	HMACSHA-256	PBKDF2WithHMACSHA-1
Motorola Moto X	HMACSHA-256	PBKDF2WithHMACSHA-1
Samsung Galaxy S3	HMACSHA-256	PBKDF2WithHMACSHA-1

Sony Xperia Tablet S	HMACSHA-256	PBKDF2WithHMACSHA-1
Sony Xperia S	HMACSHA-256	PBKDF2WithHMACSHA-1
Sony Xperia Z	HMACSHA-256	PBKDF2WithHMACSHA-1

Tabelle 1: Verfügbarkeitstests von kryptographischen Algorithmen

Auch für HMAC empfiehlt das BSI eine Mindestschlüssellänge von 128 Bit. Der aus HMAC entstehende Authentifizierungscode sollte mindestens 96 Bit lang sein. Als Grundlage für das HMAC-Verfahren dient eine Hash-Funktion. Mit *opad* und *ipad* als Konstanten, *K* als geheimen Schlüssel und *H* als geeignete Hash-Funktion, wird der MAC wie folgt berechnet:

$$MAC = H(K \oplus opad, H(K \oplus ipad, message))$$

Das BSI empfiehlt das Verwenden einer SHA-Funktion [16], die einen Wert von mindestens 256 Bit berechnet, z. B. SHA-256, SHA-384 oder SHA-512. Diese Funktionen gelten als kryptographisch stark. Die Hash-Funktion SHA-1, die einen 160 Bit langen Hash-Wert berechnet, kann nicht mehr als kollisionsresistent angesehen werden [13]. Allerdings ist nach dem BSI die Verwendung von SHA-1 in Szenarien, in denen die Kollisionsresistenz keine große Rolle spielt - zum Beispiel zum Erstellen eines MACs -, noch bis zum Jahr 2018 möglich.

Die mitgelieferten JCA-Provider (s. Praxisphasenbericht Kap. 3) implementieren auf den meisten Android Geräten bisher nur PBKDFs mit SHA-1. HMAC bieten alle getesteten Provider in Kombination mit SHA-256 an (s. Tabelle 1). SecureAndroid prüft das Gerät auf Verfügbarkeit von SHA-256 und nutzt bei Nichtverfügbarkeit SHA-1, sowohl für das Berechnen von MACs als auch als Grundlage der PBKDF. Hier stehen die Geräte-Hersteller in der Pflicht, PBKDFs mit SHA-256 oder SHA-512 auf den Geräten auszuliefern.

3.3. Angriffe auf Verschlüsselungsverfahren

Folgend werden einige für diese Arbeit relevanten Angriffe auf die Sicherheit von Verschlüsselungsverfahren betrachtet. Zuerst wird die Thematik generisch betrachtet, indem ein theoretisches Kryptosystem behandelt wird. Daraufhin werden konkrete Angriffe vorgestellt. Für die formelle Betrachtungsweise wird die Definition eines Kryptosystems benötigt und wann ein solches als sicher gilt. Für eine detaillierte Erklärung der Formalisierung von Bedrohungsszenarien wird an dieser Stelle auf [10] verwiesen.

3.3.1. Generische Betrachtung eines Kryptosystems

Ein Kryptosystem besteht aus einer Menge von Klartexten, einer Menge von Chiffrentexten, einer Familie von Ver- und Entschlüsselungsfunktionen sowie einer Menge von Schlüsseln. Um Angriffe auf ein solches System unabhängig von konkreten Systembedingungen modellieren zu können, wird ein Spiel mit einem Angreifer und einem Orakel zur Hilfe herangezogen.

In diesem Spiel muss der Angreifer einen verschlüsselten Text entschlüsseln. Das Orakel ist ein formales Hilfsmittel, um Realbedingungen zu simulieren. Der Angreifer wählt zwei Klartexte p_1 und p_2 . Er übergibt die Klartexte dem Orakel, das dann zufällig auswählt, welchen der beiden Klartexte es mit einer gewählten Verschlüsselungsfunktion chiffriert. Der Angreifer erhält darauf den Chiffertext, der entweder der Chiffertext von p_1 oder p_2 ist. Wenn der Angreifer durch den Chiffertext nur einen vernachlässigbar geringen Informationsgewinn $v(k)$ erhält, der seine Chance zu erraten, welcher der beiden Klartexte verschlüsselt wurde, nicht signifikant erhöht, gilt das System als sicher im Sinne der *polynomiellen Ununterscheidbarkeit* [10]. Dies bedeutet, dass $v(k)$ eine Funktion der Schlüssellänge k ist, so dass gilt: Für jede Polynomfunktion poly gibt es ein k' mit

$$|v(k)| < |1/\text{poly}(k)|$$

für jedes $k > k'$. Die einzige Möglichkeit des Angreifers ist also zu raten und mit einer Wahrscheinlichkeit von $0.5 + v(k)$ den Klartext zu benennen. Alternativ kann er versuchen, den Verschlüsselungsschlüssel durch einen Brute-Force-Angriff zu knacken. Dies schafft er aber nur mit exponentiellem Aufwand, was für die praktische Relevanz bedeutet, dass er es gar nicht schafft [10]. Solch ein System ist nicht absolut, für praktische Zwecke aber hinreichend sicher. Dem Angreifer stehen folgende Angriffsmöglichkeiten auf das Kryptosystem zur Verfügung:

◆ Known-Ciphertext-Attacke

Das Orakel verschlüsselt zufällig ausgewählte Klartexte und stellt dem Angreifer die entsprechenden Chiffrentexte zur Verfügung. Dies entspricht unter Realbedingungen einem Angreifer, der Chiffrentexte mitliest und dann versucht, von den Chiffrentexten Rückschlüsse auf die Klartexte bzw. den geheimen Schlüssel zu ziehen. Der Known-Ciphertext-Angriff ist die schwächste aller möglichen Angriffsmöglichkeiten, da dem Angreifer ausser dem bereits verschlüsselten Datenmaterial keine weiteren Informationen zur Verfügung stehen.

◆ Known-Plaintext-Attacke

Das Orakel verschlüsselt zufällig ausgewählte Klartexte und stellt dem Angreifer sowohl die Klartexte als auch die entsprechenden Chiffrentexte zur Verfügung. Dies ist unter realen Bedingungen dann der Fall, wenn ein Angreifer den ganzen Klartext oder Teile

davon kennt. E-Mails beispielsweise fangen häufig mit Anreden an und hören mit Grußworten auf. Der Angreifer kann also versuchen, durch Analyse des Chiffretextes mit dem Wissen von Teilen des Klartextes, Rückschlüsse auf den Rest des Klartextes oder den geheimen Schlüssel zu ziehen.

◆ Chosen-Plaintext-Attacke (CPA)

Der Angreifer sendet von ihm selbst gewählte Klartexte an das Orakel und erhält die Chiffretexte zurück. Weil der Angreifer die Klartexte selber wählt, stehen ihm deutlich mehr Informationen zur Verfügung als bei einer Known-Plaintext-Attacke. Dies ist zum Beispiel bei der Signierung von Dokumenten mit einem geheimen RSA-Schlüssel möglich. Weil hier das Signieren der exakt gleiche Vorgang wie das Verschlüsseln ist, kann ein Angreifer versuchen, Dokumente vom Angriffsopfer signieren zu lassen. Er enthält somit den Chiffertext eines von ihm gewählten Klartextes und kann so versuchen, Rückschlüsse auf den geheimen Schlüssel zu ziehen. Unter anderem aus diesem Grund wird bei einer Signatur meist nur der Hash-Wert der Nachricht und nicht die Nachricht selbst signiert, d.h. verschlüsselt.

◆ Chosen-Ciphertext-Attacke (CCA)

Der Angreifer sendet Chiffretexte an das Orakel und erhält die entschlüsselten Klartexte zurück. Die CCA ist unter den hier betrachteten Attacken die gefährlichste. Ein praktisches Beispiel ist wie bei der CPA das Signieren mit einem geheimen RSA-Schlüssel. Der Angreifer verschlüsselt Klartexte mit dem öffentlichen Schlüssel des Opfers. Er lässt sich diese Chiffretexte dann von dem Opfer signieren, was bedeutet, dass die Texte entschlüsselt werden. Der Angreifer versucht so, Rückschlüsse auf den geheimen Schlüssel zu ziehen.

Im Zusammenhang der vorgestellten Angriffe ist der Begriff der *Ununterscheidbarkeit* (engl. *Indistinguishability*) von zentraler Bedeutung. Chiffren werden in *Ununterscheidbarkeits-Kategorien* eingeteilt. Wenn eine Chiffre einer CPA standhält, gehört sie in die Kategorie *Ununterscheidbar während einer CPA* (engl. *IND-CPA*). Wenn sie einer CCA standhält, gehört die Chiffre in die Kategorie *Ununterscheidbar während einer CCA* (engl. *IND-CCA*) [9].

Neben dem Verschlüsseln von Daten prüft SecureAndroid auch deren Integrität. Daher muss neben der Ununterscheidbarkeit noch der Begriff der *Nicht-Veränderbarkeit* (engl. Non-Malleability, NM) eingeführt werden. Ein Verschlüsselungsverfahren gilt als veränderbar, wenn es einem Angreifer gelingt, einen gegebenen Chiffertext so zu verändern, dass die Entschlüsselung des Chiffretextes einen dem ursprünglichen Klartext ähnlichen Klartext ergibt. Angreifer könnten so Nachrichten an konkreten Stellen verändern, ohne die Nachricht dafür entschlüsseln zu müssen. Ein für SecureAndroid geeignetes Verfahren muss also unter dem stärkstmöglichen Angriff neben der Vertraulichkeit auch die

Integrität und Authentizität des Chiffretextes und daraus folgend des Klartextes sicherstellen. Insgesamt sollte ein in SecureAndroid eingesetztes Kryptosystem bezogen auf die Vertraulichkeit folgende Eigenschaften besitzen:

- ◆ Ununterscheidbarkeit während einer CPA
- ◆ Ununterscheidbarkeit während einer CCA

Die Ununterscheidbarkeit während eines Known-Plaintext- und Known-Ciphertext-Angriffs folgt dabei aus der Ununterscheidbarkeit während einer CCA/CPA.

Bezogen auf die Integrität und Authentizität sollte das Kryptosystem folgende Eigenschaft besitzen:

- ◆ Nicht-Veränderbarkeit während einer CPA (engl. NM-CPA)
- ◆ Integrität des Klartextes (engl. INT-PTXT)
- ◆ Integrität des Chiffretextes (engl. INT-CTXT)
- ◆ Authentizität des Chiffretextes
- ◆ Authentizität des Klartextes

Die Integrität und Authentizität des Klartextes folgt dabei aus der Integrität und Authentizität des Chiffretextes. Bei der Kombination von Verschlüsselung und dem Erstellen des MACs gibt es drei mögliche Vorgehensweisen [9]:

1. Encrypt-and-MAC

Sowohl die Verschlüsselungsoperation als auch die Berechnung des MACs geschieht auf dem Klartext. Der Chiffretext und der MAC werden zu einem Paket zusammengefügt. Zur Überprüfung der Integrität wird der Chiffretext-Anteil entschlüsselt, anhand des Klartextes der MAC berechnet und mit dem bereits berechneten MAC verglichen.

2. MAC-and-Encrypt

Zuerst wird der MAC auf dem Klartext berechnet. Der Klartext und MAC werden dann verschlüsselt und zu einem Paket zusammengefügt. Zur Überprüfung wird der Chiffretext, der aus verschlüsseltem Klartext und MAC besteht, entschlüsselt. Der MAC des entschlüsselten Klartexts wird berechnet und mit dem bereits berechneten MAC verglichen.

3. Encrypt-then-MAC

Zuerst wird der Klartext verschlüsselt. Von dem resultierenden Chiffretext wird der MAC berechnet und Chiffretext sowie MAC werden zu einem Paket zusammengefügt. Um die Integrität zu überprüfen, wird der MAC des Chiffretextes berechnet und mit dem vorher berechneten MAC verglichen. Sind beide MACs gleich, ist die Integrität der Daten sichergestellt und die Entschlüsselung kann vorgenommen werden.

Tabelle 2 zeigt eine Vergleichsübersicht der Sicherheitseigenschaften der drei Kompositionsmöglichkeiten hinsichtlich der soeben beschriebenen Kategorien.

Komposition	IND-CPA	IND-CCA	NM-CPA	INT-PTXT	INT-CTXT
Encrypt-and-MAC	o	o	o	x	o
MAC-then-Encrypt	x	o	o	x	o
Encrypt-then-MAC	x	x	x	x	x

Tabelle 2: Übersicht Kompositionsmöglichkeiten Verschlüsselung und Generierung MAC

Von diesen drei Kompositionen bietet nur die Kombination Encrypt-then-MAC Vertraulichkeit des Klartextes, Integrität des Chiffertextes und daraus folgend auch Integrität und Authenzität des Klartextes. In SecureAndroid wird deshalb nach dem Prinzip Encrypt-then-MAC vorgegangen. Nach Verschlüsselung mit AES-CBC, was IND-CPA sicherstellt, wird die HMAC-Funktion auf den Chiffertext angewendet. Dadurch werden IND-CCA, NM-CPA, INT-PTXT, INT-CTXT sowie die Authentizität der Daten sichergestellt.

3.3.2. Konkrete Angriffe auf Verfahren

Nach der formellen Betrachtung der Sicherheit eines Kryptosystems werden nun die für diese Arbeit relevanten konkreten Angriffe und Gegenmaßnahmen erläutert.

- ◆ **Wörterbuchangriff** (engl. *dictionary attack*) [16]

Bei einem Wörterbuchangriff wird versucht, ein Passwort durch das Ausprobieren von Wortkombinationen aus einem Wörterbuch zu erraten. Die meisten Passwörter enthalten tatsächlich vorhandene Wörter und sind somit anfällig für solch eine Attacke [32]. Hier kommt die PBKDF zum Tragen. Wenn diese verwendet wird, um aus Passwörtern Schlüssel zu generieren oder um Hash-Werte von Passwörtern zu erstellen, muss der Angreifer auch seine potentiellen Passwörter der PBKDF übergeben. Wegen des Work-Factors (s. Kap. 3.2) wird sein Angriff verlangsamt. Angenommen ein Angreifer kann mit einer speziellen Hardware 300.000 SHA1-Hashes pro Sekunde berechnen. Ein Work-Factor von 65.000 bewirkt, dass der Angreifer nur noch $300.000 / 65.000 = 4.6$ Hashes/sec berechnen kann. Ein solcher Work-Factor erhöht die Schlüsselstärke folgerichtig um etwa 16 Bit. Der Wörterbuchangriff unterscheidet sich von der Brute-Force-Methode darin, dass nicht alle möglichen, sondern nur durch das Wörterbuch definierte Zeichenkombinationen ausprobiert werden. Diese Wörterbücher sind meist multilin-

gual und beispielsweise an den Duden oder ähnliche Werke angelehnt. Auch themen-spezifische Passwortlisten wie Namen und Begriffe aus der Fernsehserie *Star Trek* kursieren in Internet-Tauschbörsen.

- ◆ **Rainbowtables [16]**

Rainbowtables sind berechnete Tabellen mit Hash-Werten von Passwörtern. Die Tabellen werden meist verwendet, um aus gespeicherten Hash-Werten die Klartext-Passwörter zu extrahieren. Dazu werden im Voraus Hash-Werte von Passwörtern berechnet und mit den zugehörigen Passwörtern in einer Tabelle gespeichert. Wenn ein Angreifer die gespeicherten Hash-Werte von Passwörtern aus einer Datenbank erbeuten kann, vergleicht er diese Werte mit den im Voraus berechneten. Findet er eine Übereinstimmung, hat er das Klartext-Passwort gefunden. Schutz vor diesem Angriff bietet der in Kapitel 3.2 erläuterte Salt-Wert. Wenn jedes Passwort vor Berechnung des Hash-Werts mit einem einzigartigen, zufällig generierten Salt konateniert wird, hat die Rainbowtable keinen Nutzen mehr. Bei einem 512 Bit langen Salt müsste der Angreifer jetzt für *jedes einzelne Passwort* eine 2^{512} große Tabelle berechnen.

- ◆ **Timing-Angriff [2]**

Bei einem Timing-Angriff versucht ein Angreifer durch das Messen der Dauer von kryptographischen Operationen Rückschlüsse auf den Input der Operationen zu schließen. In SecureAndroid ist dies bei dem Vergleichen von MACs relevant. Wenn zu diesem Zweck die vom Oracle-Java-SDK bereitgestellte Methode `Arrays.equals` zum Vergleichen von Byte-Arrays verwendet wird, bricht die Überprüfung der Arrays auf Gleichheit bei der ersten Nichtübereinstimmung ab. Ein Angreifer kann also die 256 möglichen Werte für das erste Byte des MACs für eine von ihm gewählte Nachricht mitliefern und messen, bei welchem dieser Werte die Verifikation etwas länger dauert. Somit hat der Angreifer das erste Byte des richtigen MACs. Nun wiederholt er diesen Vorgang für die noch verbleibenden Bytes. Hat er den korrekten MAC einer Nachricht herausgefunden, wird diese Nachricht von dem Kryptosystem als verifiziert behandelt.

Zum Schutz vor diesem Angriff verwendet SecureAndroid zum Vergleich von MACs die Methode `MessageDigest isEqual` aus dem Paket `java.security`. Diese Methode hat eine feste Laufzeit und bietet somit keine Angriffsfläche für Zeit-Analysen.

3.3.3. Modellierung von Gefahren

Um Bedrohungen für eine Anwendung systematisch zu erfassen, wird in dieser Arbeit ein vereinfachtes *Threat-Model* [30] verwendet. Dies ist eine von Microsoft eingeführte Vorgehensweise, um mögliche Gefahren für eine Anwendung zu modellieren. Dazu wird anhand eines Datenflussdiagramms untersucht, an welchen Stellen eine Software Gefahr läuft, angegriffen zu werden. Für diese Arbeit werden nur Gefahren betrachtet, welche die Authentizität, Integrität und Vertraulichkeit der Daten betreffen (s. Kap.

3.1). Das Threat-Model für SecureAndroid beinhaltet eine einfache App, die Daten speichert. Die App wird unter zwei verschiedenen Voraussetzungen betrachtet:

1. App ohne kryptographische Maßnahmen

Es wird analysiert, welche Angriffsvektoren für das Auslesen und Verändern von Daten bestehen, wenn eine App keine kryptographischen Maßnahmen zum Schutz dieser Daten vorsieht. Für diese Angriffe werden dann geeignete kryptographische Gegenmaßnahmen definiert.

2. App mit kryptographischen Maßnahmen

Nachdem die App mit grundlegenden kryptographischen Gegenmaßnahmen ausgestattet wurde, werden diese Gegenmaßnahmen auf Schwachstellen untersucht. Dazu wird insbesondere darauf geachtet, dass die in Kap. 2.2 aufgezeigten Fehler ausgeschlossen werden.

Die Bedrohungsanalyse beinhaltet das Bewerten der identifizierten Gefahren. Dazu wird das DREAD-Modell [30] verwendet. In diesem Modell werden Gefahren anhand von fünf Kategorien gewichtet:

- ◆ Damage Potential (dt. Schadenspotential)
- ◆ Reproducibility (dt. Reproduzierbarkeit)
- ◆ Exploitability (dt. Ausnutzbarkeit)
- ◆ Affected Users (dt. Betroffene Nutzer)
- ◆ Discoverability (dt. Entdeckungspotential)

Den Kategorien wird ein Wert von 0 - 3 zugewiesen. Je größer der Wert ist, desto höher ist das Bedrohungspotential. Dann werden die Werte summiert und eine Gefahr erhält eine *Dreadsum* (Bedrohungssumme). Dies ist eine Hilfe, um Gegenmaßnahmen für die identifizierten Bedrohungen zu priorisieren. Bei einem komplexen Softwaresystem geht es nicht darum, absolute Sicherheit zu schaffen. Dies ist aus den folgenden Gründen nicht möglich:

1. Fehler in Software aufgrund von hoher Komplexität

Nach [23] enthalten 1000 Zeilen Code im Mittel 15-50 Fehler (Industrie Standard). Dieser Wert zeigt auf, dass es keine fehlerfreie Software gibt. Einzig in Bereichen, wo es um nationale Sicherheit oder sehr hohe Geldsummen geht, wird Software so gründlich getestet, dass sie nahezu fehlerfrei ist. So erreichte die Nasa zwischenzeitlich 0 Fehler in 500.000 Zeilen Code für das Space-Shuttle-Projekt [23]. Allerdings geschehen auch in der Raumfahrt Fehler. Beispielsweise ist die Ariane-5 Rakete wegen eines Integer-Overflows abgestürzt [6].

2. Kompromisse aus Benutzbarkeit und Sicherheit müssen gefunden werden

Erhöhte Sicherheit bedeutet auch umständlichere Benutzbarkeit (s. Kap. 3.2). Eine hohe Iterationsanzahl bei PBKDFs bedeutet eine längere Wartezeit, ein langes Passwort muss mühsam eingegeben werden usw. Dies hat zur Folge, dass es für Programme, welche von Endanwendern genutzt werden, nie vollkommene Sicherheit geben kann. Einzig bei sicherheitskritischer Software (z. B. für das Militär) wird die Sicherheit als sehr viel wichtiger als die Benutzererfahrung gewertet.

Abb. 2 zeigt das für SecureAndroid modellierte Datenflussdiagramm. Die Tabelle mit den identifizierten Gefahren und den definierten Gegenmaßnahmen findet sich in Anhang A1.

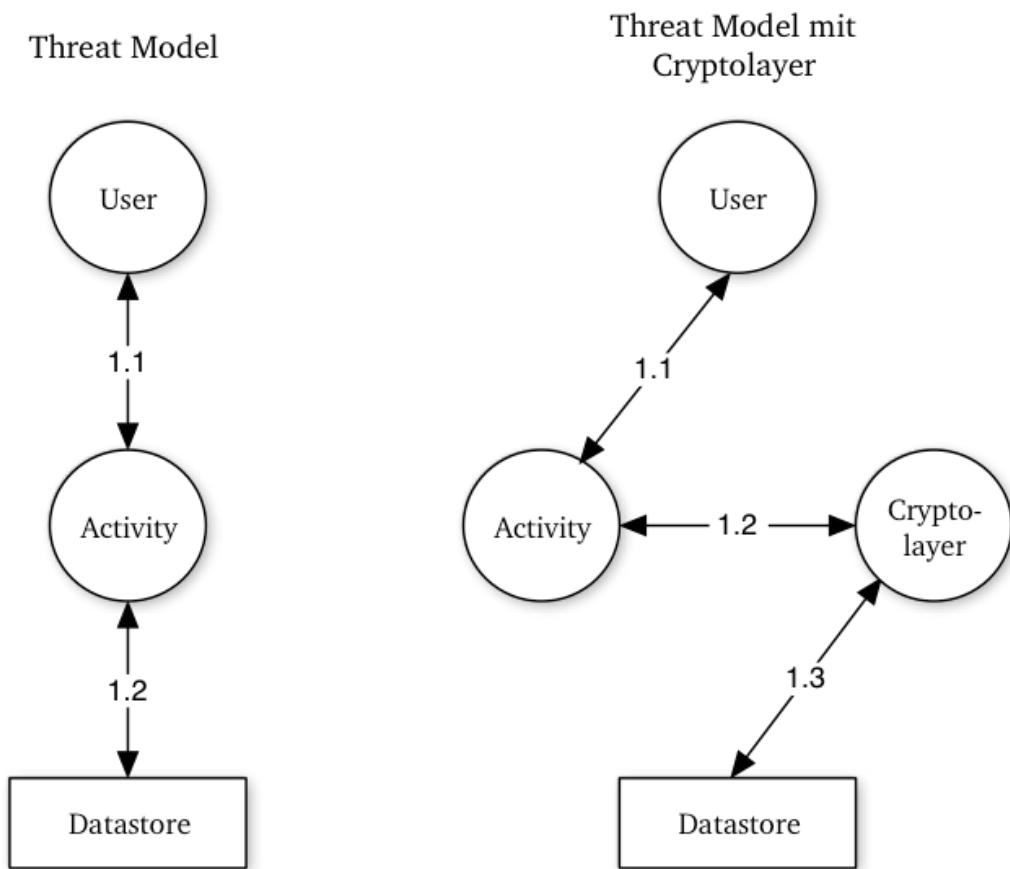


Abbildung 2: Datenflussdiagramm Threat-Model

SecureAndroid ist darauf ausgerichtet, einen guten Kompromiss zwischen Sicherheit und Benutzbarkeit zu finden. Die identifizierten Gefahren können in manchen Fällen ganz beseitigt werden, beispielsweise bei der Auswahl eines sicheren Betriebsmodus für AES. In anderen Fällen, z. B. bei den PBKDF-Iterationen, kann es nur einen Kompromiss geben.

4. Realisierung

In diesem Kapitel wird das SecureAndroid-Framework vorgestellt und dessen Funktionsweise detailliert erläutert. Zu Beginn wird ein Überblick über das Zusammenwirken der einzelnen Klassen anhand von UML-Diagrammen vermittelt. In Kapitel 4.2 wird die API vorgestellt. Kapitel 4.3 und 4.4 erläutern die Ver- und Entschlüsselungsvorgänge mitsamt der Schlüsselverwaltung anhand von Programmcode und Diagrammen. In Kapitel 4.5 werden die Methoden zum Speichern und Laden vorgestellt. Das Kapitel 4.6 behandelt das Testen von SecureAndroid und in Kapitel 4.7 wird die Nutzung in den eap-Apps erläutert.

4.1. Überblick Framework

SecureAndroid besteht aus den fünf Klassen `CryptoIOHelper`, `PasswordCrypto`, `MACCrypto`, `AESCrypto` und `SecureAndroid`. Abb. 3 zeigt ein UML-Übersichtsdiagramm des Frameworks. Die Klassen sind nach logisch in sich abgeschlossenen Funktionalitäten konzipiert.

Die Klasse `CryptoIOHelper` enthält Hilfsmethoden, welche von allen anderen Klassen benötigt werden. Dazu gehören in erster Linie I/O-Operationen sowie eine Funktion, die ein Byte-Array mit Zufallswerten füllt. Die Klassen `PasswordCrypto`, `MACCrypto` und `AESCrypto` erben von `CryptoIOHelper`, um ihre eigenen I/O-Operationen zu realisieren und Zufallszahlen zu erzeugen. Diese Klassen beinhalten Methoden zum Hashen und Speichern von Passwörtern, zum Erstellen von MACs sowie zum Ver- und Entschlüsseln mit AES. Die Klasse `SecureAndroid` definiert die API des Frameworks. Sie implementiert die Methoden, die ein Entwickler aufrufen kann und nutzt für die Realisierung ihrer Funktionalitäten die vier anderen Klassen. Nachfolgend werden die Klassen im Detail vorgestellt.

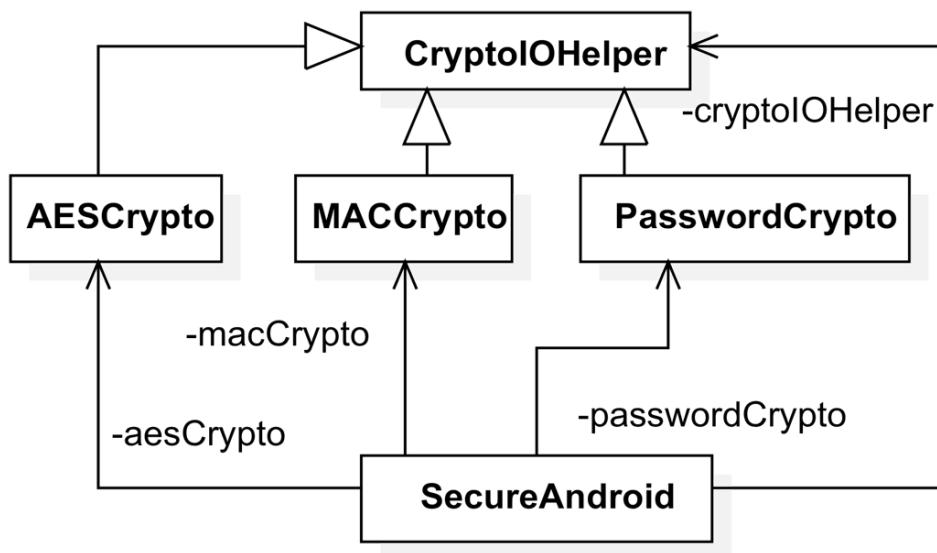


Abbildung 3: Überblick SecureAndroid

◆ CryptoIOHelper

Die Klasse `CryptoIOHelper` ist die Basisklasse der drei Krypto-Klassen und beinhaltet hauptsächlich Methoden zum Speichern und Laden von Daten. Weiter kann mit der Methode `providerCheck` überprüft werden, welche kryptographischen Algorithmen auf einem Gerät verfügbar sind. `hashPerformanceTest` bestimmt die optimale Iterationsanzahl für die PBKDF, um den bestmöglichen Kompromiss zwischen Sicherheit und Performance zu erhalten (s. Kap. 3.2). Mit `generateRandomBytes` wird eine kryptographisch sichere, pseudo-zufällige Bytefolge erstellt. `getUniquePseudoID` liest Geräteinformationen aus und erstellt aus diesen einen Hash-Wert. Dieser Wert ist auf jedem Gerät einzigartig und wird bei Bedarf als Passwort für die Ver- und Entschlüsselungsmethoden genutzt (s. Kap. 4.3).

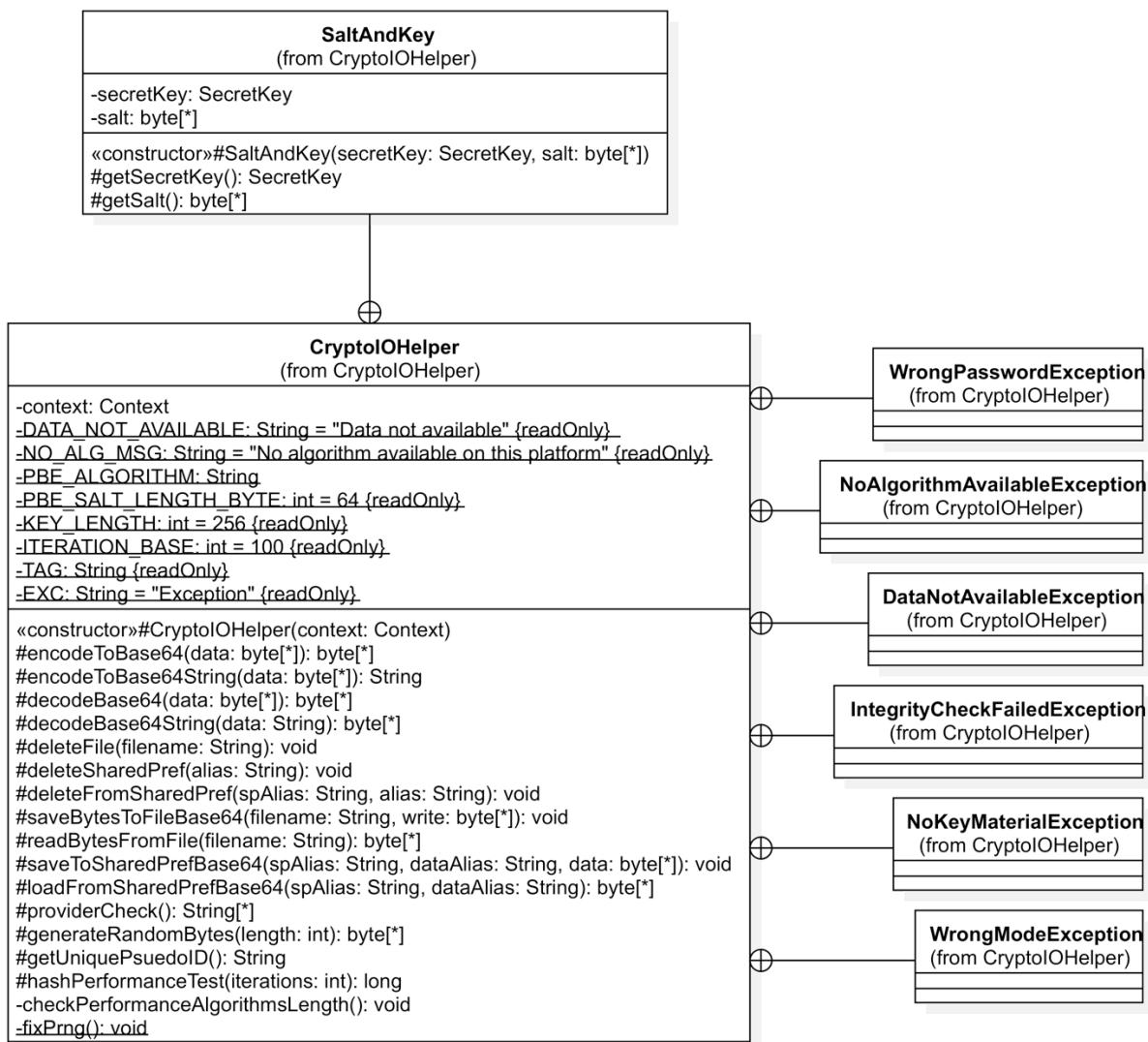


Abbildung 4: UML-Klassendiagramm von `CryptoIOHelper`

In SecureAndroid werden eigens definierte *Exceptions* zur Fehlerbehandlung verwendet. Auch diese befinden sich in `CryptoIOHelper` und können somit von allen erbenden Klassen verwendet werden. `CryptoIOHelper` beinhaltet zusätzlich die innere Klasse

`SaltAndKey`, die dafür verwendet wird, ein `SecretKey`-Objekt und den Salt, der zur Erstellung des Schlüssels verwendet wurde, zu speichern.

◆ PasswordCrypto

`PasswordCrypto` beinhaltet Methoden zum Erstellen, Speichern, Laden und Überprüfen von Passwörtern. Zum Speichern und Laden werden die Funktionen der Superklasse `CryptoIOHelper` genutzt. `PasswordCrypto` berechnet die Hash-Werte der Passwörter. Dazu wird für jedes Passwort ein zufälliger Salt-Wert generiert. Dann wird der Algorithmus HMACSHA-256 oder HMACSHA-1 auf das Passwort und den Salt angewendet. Welche der Funktionen aufgerufen wird, entscheidet sich je nach Verfügbarkeit auf dem jeweiligen Android-Gerät (s. Kap. 3.2). Zusätzlich enthält `PasswordCrypto` die innere Klasse `HashedPasswordAndSalt`. Mit dieser Hilfsklasse wird ein gehashtes Passwort mitsamt dem Salt-Wert, der in die Erstellung des Hash-Werts eingeflossen ist, gespeichert.

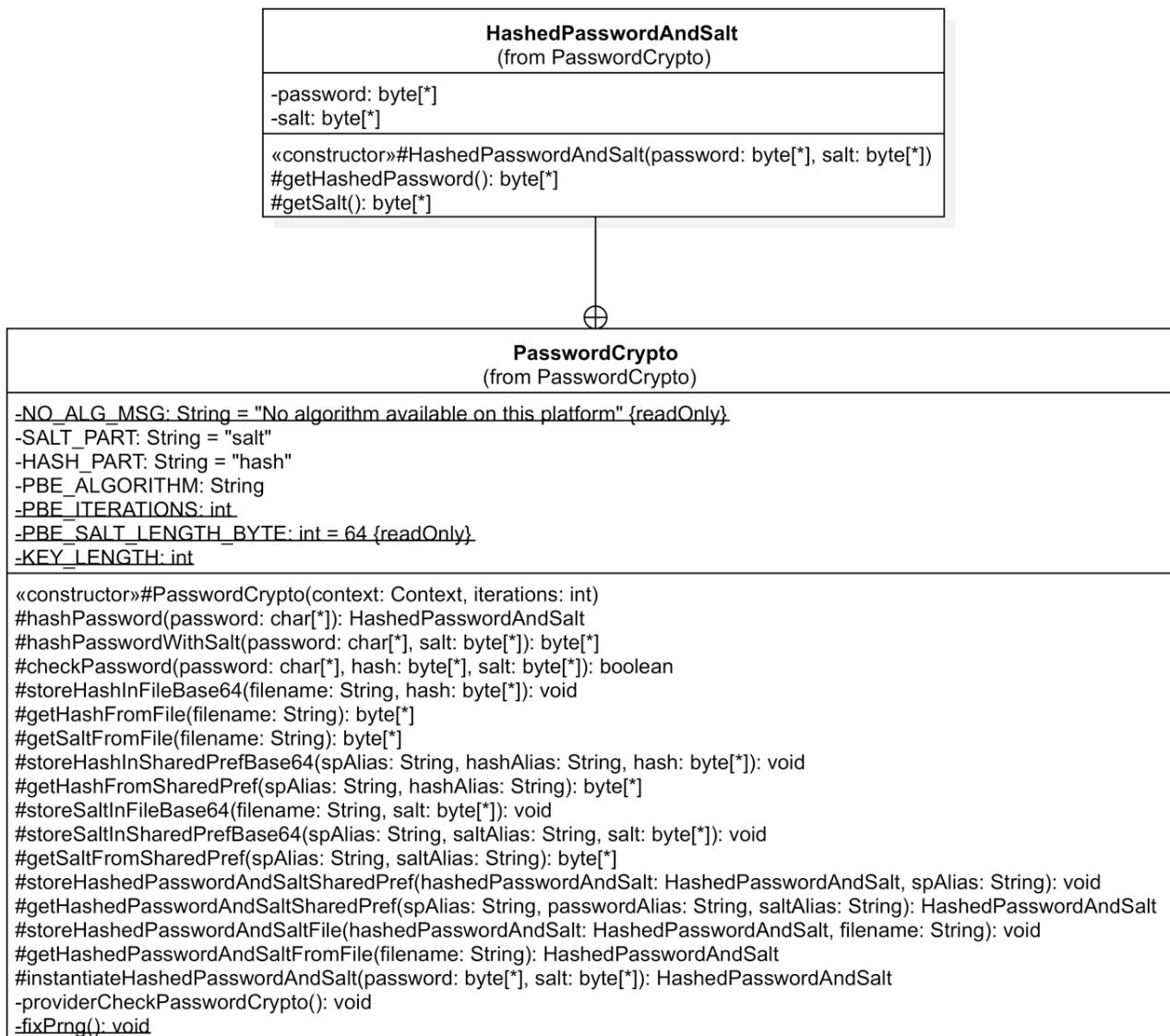


Abbildung 5: UML-Klassendiagramm von `PasswordCrypto`

◆ AESCrypto

Die Klasse AESCrypto ist für alle AES-Operationen zuständig. Dazu gehört die Schlüsselgenerierung sowie das Ver- und Entschlüsseln von Daten. Zusätzlich bietet AESCrypto die Möglichkeit, Chiffretexte mit allen zum Entschlüsseln benötigten Informationen zu speichern. Bei der Schlüsselgenerierung gibt es zwei grundsätzliche Möglichkeiten:

1. Zufällige Schlüsselgenerierung
2. Schlüsselgenerierung aus einem Passwort mittels einer PBKDF

AESCrypto enthält die innere Klasse CipherIV. CipherIV wird benötigt, um einen Chiffretext und den zugehörigen IV zu speichern. Um CipherIV auch außerhalb von AESCrypto initialisieren zu können, liefert die Methode instantiateCipherIV eine CipherIV-Instanz.

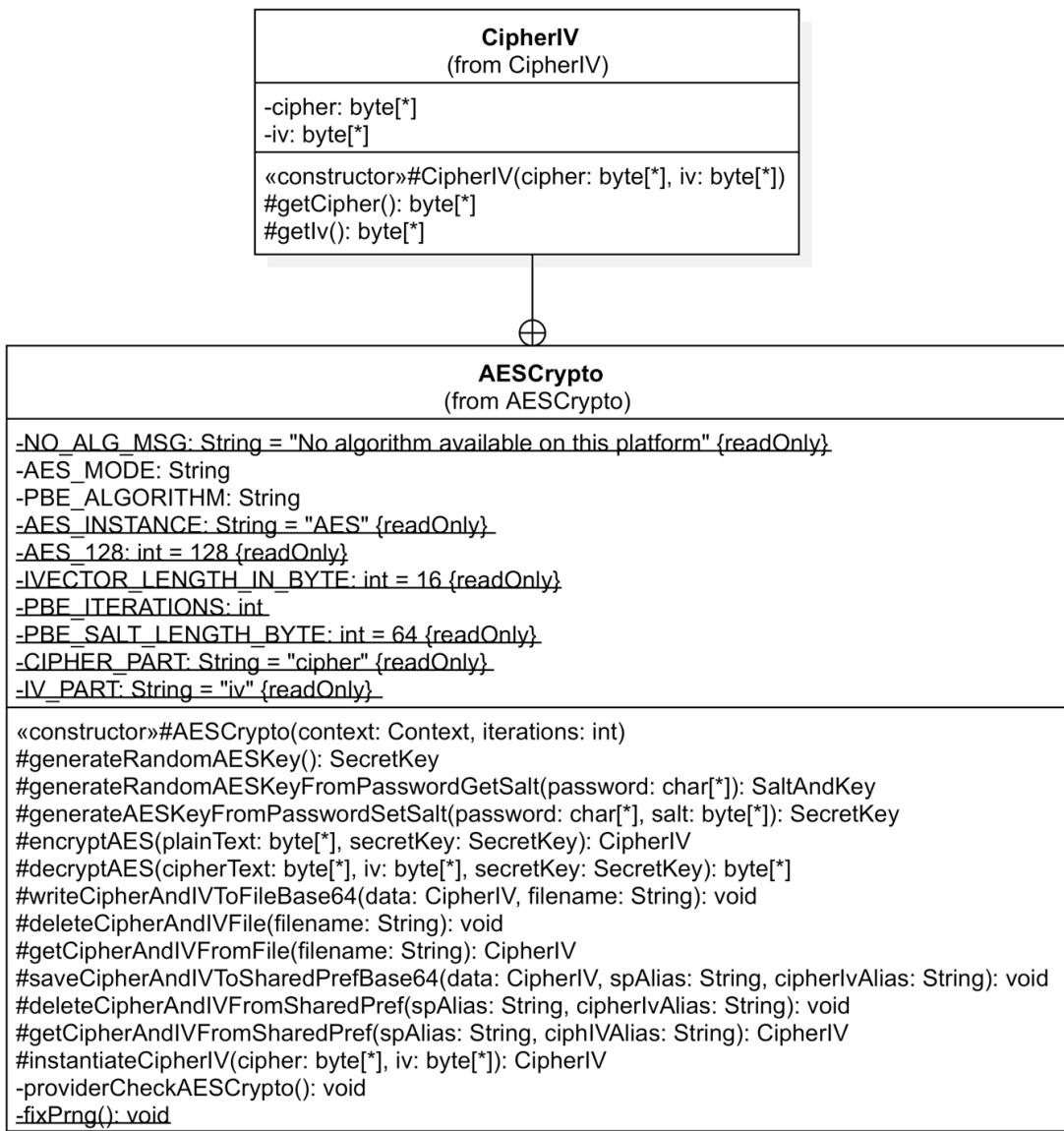


Abbildung 6: UML-Klassendiagramm von AESCrypto

◆ MACCrypto

MACCrypto beinhaltet alle zum Erstellen, Speichern, Laden und Überprüfen von MACs benötigten Methoden. Wie mit AESCrypto kann auch mit MACCrypto ein MAC-Schlüssel rein zufällig oder aus einem Passwort generiert werden. Die Methode `checkIntegrity` überprüft die Integrität eines übergebenen Byte-Arrays mit dem übergebenen Secret-Key.

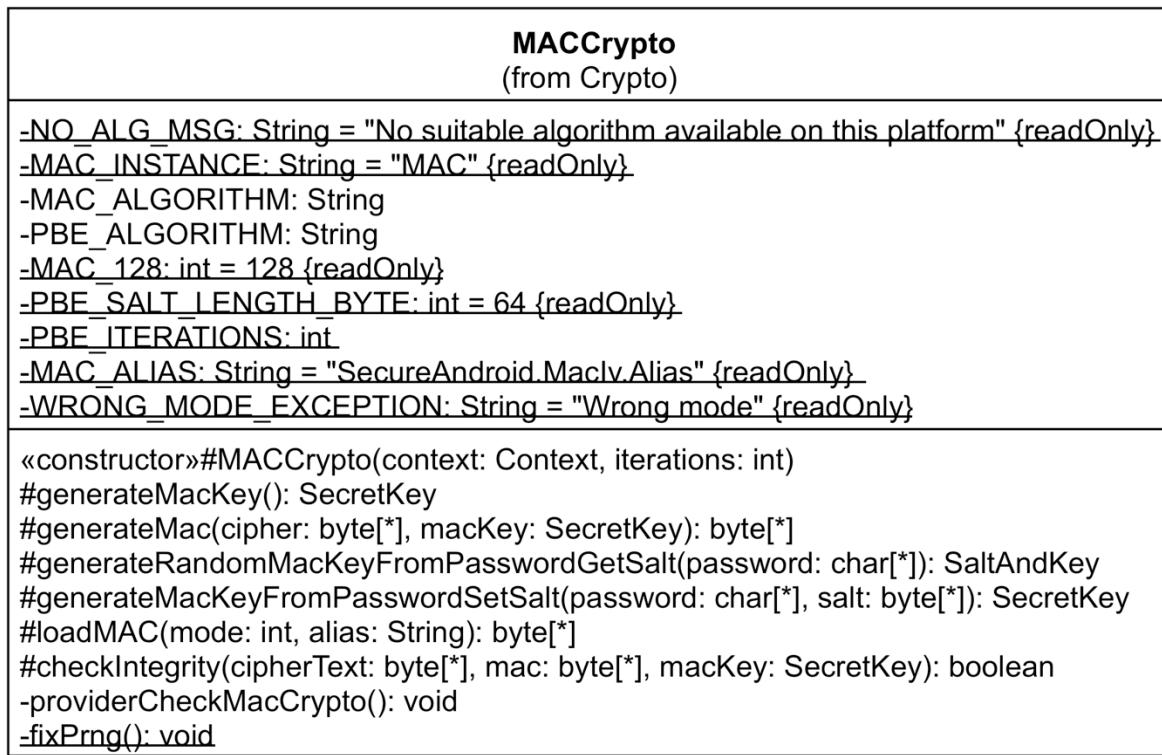


Abbildung 7: UML-Klassendiagramm von MACCrypto

◆ SecureAndroid

SecureAndroid ist die Klasse, die ein Entwickler nutzt. Sie enthält alle öffentlichen Methoden und definiert über diese die API des Frameworks. SecureAndroid besitzt Objektinstanzen aller Krypto-Klassen und von `CryptoIOHelper`. Dabei nutzt sie deren Funktionalitäten kombinatorisch so, dass die Anforderungen an das Framework (s. Kap. 1.2) erfüllt werden. Durch den modularen Aufbau der Klassen sind auch anders gestaltete APIs denkbar. Je nach Anforderung kann SecureAndroid die Methoden der Krypto-Klassen so kombinieren, dass die jeweils definierten API-Verträge erfüllt werden. Die Methoden von SecureAndroid werden in den Kapiteln 4.2 - 4.5 im Detail vorgestellt.



Abbildung 8: UML-Klassendiagramm von SecureAndroid

Zusätzlich zu den soeben vorgestellten Klassen beinhaltet `SecureAndroid` die Klasse `PRNGFixes`. `PRNGFixes` ist eine von Google bereitgestellte Klasse zur Behebung eines Fehlers im CSPRNG der Android-Versionen 4.1 - 4.3 [5]. Der Fehler besteht darin, dass der CSPRNG keine Entropie-Quelle als Seed benutzt. Auf Linux-Systemen dient `/dev/random` bzw. `/dev/urandom` als Entropie-Quelle. Auch Android nutzt diese Quellen. In den Android-Versionen 4.1 - 4.3 funktioniert dies aber nicht und der Zufallszahlengenerator liefert vorhersagbare Zufallszahlen. Da für wichtige kryptographische Operationen wie Schlüsselgenerierung, Generierung von IVs und Generierung von Salt-

Werten kryptographisch starke Zufallszahlen benötigt werden (s. Kap. 3), stellt dies ein ernsthaftes Problem dar. Der PRNG-Fehler führte im Jahr 2013 dazu, dass Bitcoins³ von Android-Geräten gestohlen wurden [31]. PRNGFixes enthält eine Methode zum Beheben des Fehlers. Diese Methode wird immer ausgeführt, bevor ein CSPRNG benutzt wird. Der Fix wird nur ausgeführt, wenn das entsprechende Gerät mit der Android-Version 4.1, 4.2 oder 4.3 läuft.

4.2. API

Die API von SecureAndroid besteht aus dem Konstruktor und 13 Methoden (s. Listing 1).

- **public** SecureAndroid(Context context, **int** minIterations)
- **public byte[]** encrypt(Integrity integrity, **byte[]** plaintext)
- **public byte[]** decrypt(Integrity integrity, **byte[]** ciphertext)
- **public byte[]** encryptWithPassword(Integrity integrity, **byte[]** plaintext, **char[]** password)
- **public byte[]** decryptWithPassword(Integrity integrity, **byte[]** ciphertext, **char[]** password)
- **public void** encryptAndStore(Integrity integrity, Mode mode, **byte[]** plaintext, String alias)
- **public void** encryptAndStoreWithPassword(Integrity integrity, Mode mode, **byte[]** plaintext, String alias, **char[]** password)
- **public byte[]** retrieve(Integrity integrity, Mode mode, String alias)
- **public byte[]** retrieveWithPassword(Integrity integrity, Mode mode, String alias, **char[]** password)
- **public void** changePassword(**char []** oldPassword, **char []** newPassword)
- **public boolean** changeFromAutoPassword(**char []** newPassword)
- **public boolean** changeToAutoPassword(**char []** oldPassword)
- **public void** deleteData(Mode mode, String alias)
- **public void** wipeKey()

Listing 1: SecureAndroid-API

Nachfolgend werden der Konstruktor und die öffentlichen Methoden behandelt. Die detaillierte Beschreibung der Ver- und Entschlüsselungsvorgänge folgt in den Kapiteln 4.3. und 4.4.

◆ SecureAndroid-Konstruktor

SecureAndroid wird als Java-Objekt initialisiert und nimmt eine Context-Variable [4] sowie eine Mindestanzahl an Iterationen für die PBKDF entgegen (s. Listing 2).

³ <http://www.coin-desk.com/information/what-is-bitcoin/>

```

/**
 * The SecureAndroid constructor.
 *
 * @param context The context.
 * @param minIterations The minimum iterations for the PBKDF.
 * @throws CryptoIOHelper.NoAlgorithmAvailableException
 * @throws NoSuchAlgorithmException
 * @throws InvalidKeySpecException
 */
public SecureAndroid(Context context, int minIterations) throws
    CryptoIOHelper.NoAlgorithmAvailableException, NoSuchAlgorithmException,
    InvalidKeySpecException {
    // Instantiate CryptoIOHelper
    cryptoIOHelper = new CryptoIOHelper(context);
    // Prepare iterations
    int iterations;
    // Check if performance-test already happened and load iterations
    try {
        iterations = Integer.parseInt(new String(cryptoIOHelper
            .loadFromSharedPrefBase64(ITERATION_COUNT_ALIAS,
            ITERATION_COUNT_ALIAS)));
    } catch (CryptoIOHelper.DataNotAvailableException e) {
        // If not, do performance-test
        // Get the suitable iteration count for good performance/security
        iterations = (int)(cryptoIOHelper.hashPerformanceTest(
            ITERATION_MIDDLE, minIterations))/ITERATION_FACTOR;
        // Save the iteration count
        cryptoIOHelper.saveToSharedPrefBase64(ITERATION_COUNT_ALIAS,
            ITERATION_COUNT_ALIAS, String.valueOf(iterations).getBytes());
    }
    // Instantiate Crypto-classes
    aesCrypto = new AESCrypto(context, iterations);
    passwordCrypto = new PasswordCrypto(context, iterations);
    macCrypto = new MACCrypto(context, iterations);
    // Check and define the MAC length according to the availability of
    // SHA256/1 on the current platform
    checkMACLength();
    MACPLUSIV_LENGTH_BYTE = IV_LENGTH_BYTE+MAC_LENGTH_BYTE;
    // Apply PRNGFIX
    fixPrng();
}

```

Listing 2: SecureAndroid Konstruktor

Das Context-Objekt enthält Informationen über Parameter der derzeit aktiven App und wird benötigt, um auf App-spezifische Ressourcen wie den für die App bereitgestellten Bereich des Dateisystems und die SharedPreferences zuzugreifen (s. Praxisphasenbericht Kap. 4.4). Der Context ist für den Entwickler in jeder Activity bzw. jedem Fragment - dies sind die Grundbausteine jeder Android-App [1] - abrufbar. Somit kann SecureAndroid an jeder Stelle einer Android-App verwendet werden. SecureAndroid

besitzt Instanzen der Klassen `CryptoIOHelper`, `AESCrypto`, `MACCrypto` und `PasswordCrypto`, die mit dem übergebenen Context und den Iterationen initialisiert werden. SecureAndroid arbeitet bei der Schlüsselgenerierung sowie Ver- und Entschlüsselung nicht streng nach objektorientierten Regeln. Bei jedem Ver- und Entschlüsselungsvorgang werden die AES- und MAC-Master-Keys generiert, die AES- und MAC-Intermediate-Keys geladen, entschlüsselt und dann eingesetzt. Das entschlüsselte Schlüsselmaterial liegt somit ausschließlich zur Laufzeit der Methode im Arbeitsspeicher. Dadurch wird die Wahrscheinlichkeit, dass ein Angreifer Schlüsselmaterial im Klartext aus dem Arbeitsspeicher auslesen kann, verringert (s. Threat-Model Anhang A1). Durch die Design-Entscheidung, Schlüsselmaterial nicht global als Objektvariable verfügbar zu halten, leidet bei der Ver- und Entschlüsselung die Performance. Wie in Kapitel 3.2 erläutert, muss bei der Iterationsanzahl der wiederholt angewendeten Hash-Funktion im Rahmen der PBKDF ein Kompromiss aus Nutzbarkeit und Sicherheit gefunden werden. Deshalb ruft der `SecureAndroid`-Konstruktor nach Initialisierung von `CryptoIOHelper` eine Methode zum Testen der Performance auf (s. Listing 3).

```
iterations = (int)(cryptoIOHelper.hashPerformanceTest(ITERATION_MIDDLE,
minIterations))/ITERATION_FACTOR;
```

Listing 3: Performance-Test

Die Methode `hashPerformanceTest` (s. Anhang B9) führt die Hash-Funktion 10.000 (**ITERATION_MIDDLE**) mal aus und misst die Zeit, die das Smartphone für den Vorgang braucht. Der Kehrwert dieses Ergebnis wird als Multiplikationsfaktor für das Berechnen der Iterationsanzahl genommen. Umso kleiner also der gemessene Zeitunterschied ist, desto größer ist die Iterationsanzahl. Die Formel zur Berechnung ist darauf ausgelegt, die Anzahl an Iterationen zu finden, die auf dem jeweiligen Gerät bei einem Ver- bzw. Entschlüsselungsvorgang zu einer Wartezeit von ~500ms führt. Die Berechnung erfolgt also nicht in iterativen Schritten, weil dies zu lange dauern würde. Die Untergrenze für die Iterationen kann dem Konstruktor von `SecureAndroid` übergeben werden, beträgt aber mindestens 1000. Durch diesen Ansatz ist `SecureAndroid` zukunftsfähig. Die Iterationsanzahl skaliert bei leistungsfähigen Smartphones nach oben. Die Iterationen werden den Konstruktoren der Krypto-Klassen als Argument übergeben und in den Shared-Preferences gespeichert um bei der nächsten Initialisierung von `SecureAndroid` geladen werden zu können. Der Performance-Test findet also nur beim ersten Benutzen von `SecureAndroid` im Context einer App statt. Danach ist die optimale Iterationsanzahl gespeichert und muss nicht mehr berechnet werden. Nach diesem Vorgang ruft der Konstruktor `checkMACLength()` auf. Diese Methode überprüft, ob auf dem jeweiligen Gerät der Algorithmus HMACSHA-256 oder HMACSHA-1 vorhanden ist und passt die Länge von möglichen MACs an. Diese Information wird später bei der Ver- und Entschlüsselung benötigt.

◆ **encrypt**

Die öffentliche encrypt-Methode (s. Listing 4) dient zum Verschlüsseln von Daten in Form eines Byte-Arrays und liefert die verschlüsselten Daten wiederum als Byte-Array zurück. Der Entwickler ist bei dieser Methode frei in der Weiterverarbeitung der verschlüsselten Daten.

```
/*
 * Encrypts the given byte array plaintext and returns the ciphertext as
 * byte array.
 *
 * @param integrity The integrity mode. Choose SecureAndroid.Integrity
 * .INTEGRITY or NO_INTEGRITY.
 * @param plaintext The plaintext as byte array.
 * @return The ciphertext as byte array.
 * @throws CryptoI0Helper.NoKeyMaterialException
 * @throws CryptoI0Helper.IntegrityCheckFailedException
 * @throws GeneralSecurityException
 * @throws CryptoI0Helper.DataNotAvailableException
 * @throws CryptoI0Helper.WrongPasswordException
 */
public byte[] encrypt(Integrity integrity, byte[] plaintext) throws
    CryptoI0Helper.NoKeyMaterialException, CryptoI0Helper
    .IntegrityCheckFailedException, GeneralSecurityException,
    CryptoI0Helper.WrongPasswordException, CryptoI0Helper
    .DataNotAvailableException {
    return encrypt(integrity, plaintext, getAutoPassword().toCharArray());
}
```

Listing 4: Öffentliche encrypt-Methode

Das von `getAutoPassword()` generierte Passwort ist ein Hash-Wert von Geräte-spezifischen Informationen und somit auf jedem Smartphone einzigartig. Aus diesem Passwort werden die AES- und MAC-Master-Keys erstellt. Während diese Vorgehensweise nicht absolut sicher ist, weil auch ein Angreifer diese Informationen auslesen kann, bietet sie doch einen guten Standardschutz vor unspezifischen Angriffen und genügt deutschen Datenschutzanforderungen. Ein Angreifer müsste ganz konkret an von Secure-Android gesicherten Daten interessiert sein, um einen Angriff mittels Auslesen der von `getAutoPassword()` herangezogenen Informationen zu starten. Ein Entwickler hat mit dieser Methode die Möglichkeit, ohne über Schlüsselverwaltung und Passwörter nachdenken zu müssen, deutschen Gesetzesanforderungen zu genügen und seine bzw. die Daten seiner Kunden gegen allgemeine Angriffe zu schützen.

Weiter wird der encrypt-Methode ein Integrity-Flag vom Typ enum übergeben. Das enum kann entweder den Wert **INTEGRITY** oder **NO_INTEGRITY** annehmen (s. Listing 5).

```
public enum Integrity {
    INTEGRITY, NO_INTEGRITY
}
```

Listing 5: Integrity-Enum

Der Entwickler kann mit diesem Flag entscheiden, ob bei einem Verschlüsselungsvorgang ein MAC des Chiffretextes berechnet wird oder nicht. Wenn die Anforderung an Integrität nicht gegeben ist, kann die rechenintensive MAC-Berechnung ausgeschaltet werden, um einen Performance-Gewinn zu erhalten. In diesem Fall werden die Daten nur verschlüsselt und die MAC-Berechnung entfällt. `encrypt` leitet den Klartext zusammen mit einem automatisch generierten Passwort an die private `encrypt`-Methode weiter. Listing 6 zeigt die Signatur dieser Methode.

```
private byte[] encrypt(Integrity integrity, byte[] plaintext, char[] pass  
word)
```

Listing 6: Signatur private encrypt-Methode

Diese Methode liefert ein Byte-Array. In diesem Array sind der Chiffretext und der dem Chiffretext zugehörige IV sowie der MAC des Chiffretexts gespeichert, falls die MAC-Berechnung angefordert wurde. Der IV und bei Bedarf der MAC werden zur Entschlüsselung benötigt. Der Verschlüsselungsvorgang wird in Kapitel 4.3 detailliert erläutert.

◆ decrypt

Die öffentliche `decrypt`-Methode (s. Listing 7) ist das Pendant zur öffentlichen `encrypt`-Methode und nimmt das von `encrypt` gelieferte Byte-Array entgegen, um es mit dem automatisch generierten Passwort wieder zu entschlüsseln.

```
/**  
 * Decrypts the given byte array ciphertext and returns the plaintext as  
 * byte array.  
 *  
 * @param integrity The integrity mode. Choose SecureAndroid.Integrity  
 * .INTEGRITY or NO_INTEGRITY.  
 * @param ciphertext The ciphertext as byte array.  
 * @return The plaintext as byte array.  
 * @throws CryptoIOHelper.NoKeyMaterialException  
 * @throws GeneralSecurityException  
 * @throws CryptoIOHelper.WrongPasswordException  
 * @throws CryptoIOHelper.DataNotAvailableException  
 * @throws CryptoIOHelper.IntegrityCheckFailedException
```

```
/*
public byte[] decrypt(Integrity integrity, byte[] ciphertext) throws
    CryptoI0Helper.NoKeyMaterialException, GeneralSecurityException,
    CryptoI0Helper.WrongPasswordException, CryptoI0Helper
    .DataNotAvailableException, CryptoI0Helper
    .IntegrityCheckFailedException {
    return decrypt(integrity, ciphertext, getAutoPassword().toCharArray());
}
```

Listing 7: Öffentliche decrypt-Methode

So wie encrypt ruft decrypt die private decrypt-Methode auf und leitet das Byte-Array sowie das Integrity-Flag weiter. Das Passwort wird äquivalent zu encrypt generiert und der privaten Methode übergeben. In der privaten decrypt-Methode (Signatur s. Listing 8) wird das Byte-Array welches den Chiffretext, den IV und bei Integritätsprüfung den MAC enthält, wieder in die einzelnen Bestandteile zerlegt und die Entschlüsselung vorgenommen. Dieser Vorgang wird in Kapitel 4.4 detailliert erläutert.

```
private byte[] decrypt(Integrity integrity, byte[] ivAndCipherAndMAC,
    char[] password)
```

Listing 8: Signatur private decrypt-Methode

◆ encryptWithPassword

encryptWithPassword ist wie die öffentliche encrypt-Methode aufgebaut und ruft die private encrypt-Methode auf. Der Unterschied zur öffentlichen encrypt-Methode besteht darin, dass der privaten encrypt-Methode das vom Entwickler bereitgestellte und nicht das automatisch generierte Passwort übergeben wird.

```
/**
 * Encrypts the given byte array plaintext with a key derived from the
 * provided password and returns the ciphertext as a byte array.
 *
 * @param integrity The integrity mode. Choose SecureAndroid.Integrity
 * .INTEGRITY or NO_INTEGRITY.
 * @param plaintext The plaintext as byte array.
 * @param password The desired password that will be used to derive the
 * encryption key.
 * @return The ciphertext as a byte array.
 * @throws CryptoI0Helper.NoKeyMaterialException
 * @throws GeneralSecurityException
 * @throws CryptoI0Helper.IntegrityCheckFailedException
 * @throws CryptoI0Helper.WrongPasswordException
 * @throws CryptoI0Helper.DataNotAvailableException
 */
public byte[] encryptWithPassword(Integrity integrity, byte[] plaintext,
```

```

char[] password) throws CryptoI0Helper.NoKeyMaterialException,
CryptoI0Helper.IntegrityCheckFailedException, GeneralSecurityException,
CryptoI0Helper.WrongPasswordException, CryptoI0Helper
    .DataNotAvailableException {
return encrypt(integrity, plaintext, password);
}

```

Listing 9: encryptWithPassword-Methode

Der Entwickler hat mit dieser Methode die Möglichkeit, ein selbst gewähltes oder vom Nutzer abgefragtes Passwort zu nutzen. Wenn das Passwort vom Nutzer eingegeben wird, erhöht dies die Sicherheit gegenüber dem automatisch generierten Passwort signifikant. Das BSI empfiehlt eine Passwortlänge von mindestens 12 Zeichen [11].

◆ decryptWithPassword

Wie die öffentliche `decrypt`-Methode ruft `decryptWithPassword` (s. Listing 10) die private `decrypt`-Methode auf und übergibt ihr anstelle des automatisch generierten Passworts das vom Nutzer oder Entwickler zur Verfügung gestellte Passwort. `decryptWithPassword` ist das Pendant zu `encryptWithPassword`.

```

/*
 * Decrypts the given byte array ciphertext with a key derived from the
 * provided password and returns the plaintext as byte array.
 *
 * @param integrity      The integrity mode. Choose SecureAndroid
 *                      .Integrity.INTEGRITY or NO_INTEGRITY.
 * @param ciphertext     The ciphertext as byte array.
 * @param password       The desired password that will be used to derive
 *                      the decryption key.
 * @return               The plaintext as a byte array.
 * @throws CryptoI0Helper.NoKeyMaterialException
 * @throws CryptoI0Helper.DataNotAvailableException
 * @throws CryptoI0Helper.WrongPasswordException
 * @throws GeneralSecurityException
 * @throws CryptoI0Helper.IntegrityCheckFailedException
 */
public byte[] decryptWithPassword(Integrity integrity, byte[] ciphertext,
    char[] password) throws CryptoI0Helper.NoKeyMaterialException,
CryptoI0Helper.DataNotAvailableException, CryptoI0Helper
    .WrongPasswordException, GeneralSecurityException, CryptoI0Helper
    .IntegrityCheckFailedException {
return decrypt(integrity, ciphertext, password);
}


```

Listing 10: decryptWithPassword-Methode

◆ **encryptAndStore**

Die Methode `encryptAndStore` (s. Listing 11) speichert die Daten nach dem Verschlüsseln. Zu Beginn der Methode wird überprüft, welcher Integritäts-Modus gewünscht ist. Dann wird die Methode `saveUserCipherMACIV` bzw. `saveUserCipherIV` zum Speichern aufgerufen.

```
/*
 * Encrypts the given plaintext and stores it on the device under the
 * provided alias.
 *
 * @param integrity      The integrity mode. Choose SecureAndroid
 * .Integrity.INTEGRITY or NO_INTEGRITY.
 * @param mode            The storage mode. Choose SecureAndroid
 * .Mode.SHARED_PREFERENCES or FILE.
 * @param plaintext       The plaintext as a byte array.
 * @param alias           The alias under which the data will be stored.
 * @throws IOException
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 * @throws CryptoIOHelper.NoKeyMaterialException
 * @throws IllegalArgumentException
 */
public void encryptAndStore(Integrity integrity, Mode mode, byte[]
    plaintext, String alias) throws CryptoIOHelper.NoKeyMaterialException,
    CryptoIOHelper.IntegrityCheckFailedException, IOException,
    CryptoIOHelper.WrongPasswordException, GeneralSecurityException,
    CryptoIOHelper.DataNotAvailableException {
    switch (integrity) {
        case INTEGRITY:
            // Save the encrypted data under the given alias
            saveUserCipherMACIV(mode, encrypt(integrity, plaintext,
                getAutoPassword().toCharArray()), alias);
            break;
        case NO_INTEGRITY:
            saveUserCipherIV(mode, encrypt(integrity, plaintext,
                getAutoPassword().toCharArray()), alias);
            break;
        default:
            throw new IllegalArgumentException(
                WRONG_INTEGRITY_MODE_EXCEPTION);
    }
}
```

Listing 11: `encryptAndStore`-Methode

`saveUserCipherMACIV`/`saveUserCipherIV` wird das Ergebnis des privaten `encrypt`-Aufrufs, ein Mode-Flag und das Alias, unter welchem die Daten abgespeichert werden

sollen, übergeben. Das Mode-Flag ist wie das Integrity-Flag vom Typ enum und kann den Wert ***SHARED_PREFERENCES*** oder ***FILE*** annehmen. Das Flag bestimmt, ob die verschlüsselten Daten in den SharedPreferences oder als Datei gespeichert werden.

- ◆ **encryptAndStoreWithPassword**

Analog zu encryptAndStore prüft encryptAndStoreWithPassword (s. Listing 12) den Integritäts-Modus, ruft dann die Methode saveUserCipherMACIV bzw. saveUserCipherIV auf und übergibt ihr das Mode-Flag, das Alias und das Ergebnis des privaten encrypt-Aufrufs. Wie bei encryptWithPassword besteht der einzige Unterschied zu encryptAndStore darin, dass der privaten encrypt-Methode ein selbst gewähltes und nicht das automatisch generierte Passwort übergeben wird.

```
/*
 * Encrypts the given plaintext and stores it on the device under the
 * provided alias with a key derived from the provided password.
 *
 * @param integrity      The integrity mode. Choose SecureAndroid
 * .Integrity.INTEGRITY or NO_INTEGRITY.
 * @param mode            The storage mode. Choose SecureAndroid
 * .Mode.SHARED_PREFERENCES or FILE.
 * @param plaintext       The plaintext as a byte array.
 * @param alias           The alias under which the data will be stored.
 * @param password        The password that will be used to derive the
 * decryption key.
 * @throws IllegalArgumentException
 * @throws IOException
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws CryptoIOHelper.NoKeyMaterialException
 */
public void encryptAndStoreWithPassword(Integrity integrity, Mode mode,
    byte[] plaintext, String alias, char[] password) throws
CryptoIOHelper.NoKeyMaterialException, CryptoIOHelper
    .IntegrityCheckFailedException, IOException, CryptoIOHelper
    .WrongPasswordException, GeneralSecurityException,
CryptoIOHelper.DataNotAvailableException {
    switch (integrity) {
        case INTEGRITY:
            // Save the encrypted data under the given alias
            saveUserCipherMACIV(mode, encrypt(integrity, plaintext,
                password), alias);
            break;
        case NO_INTEGRITY:
            // Save the encrypted data under the given alias
            saveUserCipherIV(mode, encrypt(integrity, plaintext, password),
                alias);
            break;
    }
}
```

```

default:
    throw new IllegalArgumentException(
        WRONG_INTEGRITY_MODE_EXCEPTION);
}
}

```

Listing 12: encryptAndStoreWithPassword-Methode

◆ **retrieve**

Die öffentliche `retrieve`-Methode (s. Listing 13) liefert die unter dem ihr übergebenen Alias gespeicherten Daten entschlüsselt zurück und operiert dabei mit dem automatisch generierten Passwort. Sie ruft dazu die private `retrieve`-Methode auf, die sich um das Laden und Entschlüsseln der Daten kümmert. Der privaten `retrieve`-Methode wird das `Integrity`-Flag, das `Mode`-Flag, das Daten-Alias sowie das Ergebnis von `getAutoPassword()` übergeben.

```

/*
 * Retrieves and decrypts the data stored under the provided alias.
 *
 * @param integrity The integrity mode. Choose SecureAndroid.Integrity
 * .INTEGRITY or NO_INTEGRITY.
 * @param mode      The storage mode. Choose SecureAndroid.Mode
 * .SHARED_PREFERENCES or FILE.
 * @param alias     The alias the data was stored under.
 * @return          The plaintext as a byte array.
 * @throws CryptoIOHelper.NoKeyMaterialException
 * @throws IllegalArgumentException
 * @throws IOException
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 */
public byte[] retrieve(Integrity integrity, Mode mode, String alias) throws
    CryptoIOHelper.NoKeyMaterialException, CryptoIOHelper
    .IntegrityCheckFailedException, IOException, CryptoIOHelper
    .WrongPasswordException, GeneralSecurityException,
    CryptoIOHelper.DataNotAvailableException {
    return retrieve(integrity, mode, alias, getAutoPassword()
        .toCharArray());
}


```

Listing 13: retrieve-Methode

◆ `retrieveWithPassword`

Analog zur öffentlichen `retrieve`-Methode ruft `retrieveWithPassword` (s. Listing 14) die private `retrieve`-Methode auf und übergibt ihr das Integrity- und Mode-Flag, das Alias sowie das von dem Entwickler bereitgestellte Passwort.

```
/*
 * Retrieves and decrypts data stored under the provided alias.
 *
 * @param integrity The integrity mode. Choose SecureAndroid.Integrity
 * .INTEGRITY or NO_INTEGRITY.
 * @param mode      The storage mode. Choose SecureAndroid.Mode
 * .SHARED_PREFERENCES or FILE.
 * @param alias     The alias the data was stored under.
 * @param password The password that was used for storage.
 * @return          The plaintext as a byte array.
 * @throws CryptoIOHelper.NoKeyMaterialException
 * @throws IOException
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 */
public byte[] retrieveWithPassword(Integrity integrity, Mode mode, String
    alias, char[] password) throws CryptoIOHelper.NoKeyMaterialException,
    CryptoIOHelper.IntegrityCheckFailedException, IOException,
    CryptoIOHelper.WrongPasswordException, CryptoIOHelper
    .DataNotAvailableException, GeneralSecurityException {
    return retrieve(integrity, mode, alias, password);
}
```

Listing 14: `retrieveWithPassword`-Methode

◆ `changePassword/changeFromAutoPassword/changeToAutoPassword`

Bei Nutzung von SecureAndroid wird das beim ersten Verschlüsselungsvorgang benutzte Passwort in den SharedPreferences einer App gespeichert. Dieses Passwort ist ab diesem Zeitpunkt das gültige Passwort im Applikations-Context. Wenn ein Entwickler das Passwort ändern möchte, kann er dazu die `changePassword`-Methoden (s. Listing 15) aufrufen. Die Methode `changePassword` verändert das selbst gewählte Passwort zu einem anderen selbst gewählten Passwort. Mit den Methoden `changeFromAutoPassword` und `changeToAutoPassword` kann von dem automatisch generierten auf ein selbst gewähltes Passwort und umgekehrt gewechselt werden. Die drei öffentlichen Methoden rufen jeweils die private Methode `changePasswordPrivate` (s. Anhang B5) mit den Passwort-Parametern auf. Die `changePasswordPrivate`-Methode nimmt das alte Passwort entgegen, prüft dessen Richtigkeit und erstellt im Erfolgsfall neue Master-Keys anhand des neuen Passworts. Dazu entschlüsselt die Methode die Intermediate-Keys mit dem aus dem alten Passwort generierten Master-Key. Dann werden die Intermediate-

Keys mit dem aus dem neuen Passwort generierten Master-Key wieder verschlüsselt und gespeichert.

```
/*
 * Method to change the password.
 *
 * @param oldPassword The old password.
 * @param newPassword The new password.
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 */
public boolean changePassword(char [] oldPassword, char [] newPassword)
    throws CryptoIOHelper.WrongPasswordException, CryptoIOHelper
        .IntegrityCheckFailedException, GeneralSecurityException,
    CryptoIOHelper.DataNotAvailableException {
    return changePasswordPrivate(oldPassword, newPassword);
}

/**
 * Method to change the from the auto-generated password to a chosen
 * password.
 *
 * @param newPassword The new password.
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 */
public boolean changeFromAutoPassword(char [] newPassword) throws
    CryptoIOHelper.WrongPasswordException, CryptoIOHelper
        .IntegrityCheckFailedException, GeneralSecurityException,
    CryptoIOHelper.DataNotAvailableException {
    return changePasswordPrivate(getAutoPassword().toCharArray(),
        newPassword);
}

/**
 * Method to change to the auto-generated password.
 *
 * @param oldPassword The old password.
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 */
public boolean changeToAutoPassword(char [] oldPassword) throws
    CryptoIOHelper.WrongPasswordException, CryptoIOHelper
        .IntegrityCheckFailedException, GeneralSecurityException,
```

```
CryptoIOHelper.DataNotFoundException {
    return changePasswordPrivate(oldPassword, getAutoPassword()
        .toCharArray());
}
```

Listing 15: changePassword-Methoden

◆ deleteData

`deleteData` (s. Listing 16) ist eine Methode zum Löschen von gespeicherten Daten. Ihr wird das Daten-Alias und das Mode-Flag übergeben. `deleteData` löscht daraufhin die unter dem übergebenen Alias gespeicherten Daten.

```
/**
 * Deletes either the SharedPref alias entry or the file saved under the
 * alias.
 *
 * @param mode      The storage mode. Choose SecureAndroid.Mode
 * .SHARED_PREFERENCES or FILE.
 * @param alias     The alias of the SharedPref entry or the filename.
 * @throws IllegalArgumentException
 * @throws CryptoIOHelper.DataNotFoundException
 */
public void deleteData (Mode mode, String alias) throws
CryptoIOHelper.DataNotFoundException {
switch (mode) {
    case SHARED_PREFERENCES:
        aesCrypto.deleteCipherAndIVFromSharedPref(CIPHER_IV_ALIAS,
            alias);
        break;
    case FILE:
        aesCrypto.deleteCipherAndIVFile(alias);
        break;
    default:
        throw new IllegalArgumentException(WRONG_MODE_EXCEPTION);
}
}
```

Listing 16: deleteData-Methode

◆ wipeKey

Wenn ein Entwickler einen Mechanismus zur Löschung aller Daten vorsieht, kann dazu die Methode `wipeKey` (s. Listing 17) verwendet werden. `wipeKey` löscht die Intermediate-Keys und macht somit alle Daten unbrauchbar.

```
/**
 * Wipes the intermediate key and thus destroys all formerly encrypted data
```

```

* in the sense of it being irrecoverable.
*/
public void wipeKey() {
    cryptoIOHelper.deleteSharedPref(IMEDIATE_KEY_DATA);
    cryptoIOHelper.deleteSharedPref(KEY_DATA_ALIAS);
    cryptoIOHelper.deleteSharedPref(PASSWORD_ALIAS);
}

```

Listing 17: wipeKey-Methode

4.3. Verschlüsselung

Wenn die öffentlichen Verschlüsselungs-Methoden von SecureAndroid aufgerufen werden, leiten diese den Aufruf mit den entsprechenden Parametern an die private encrypt-Methode weiter (s. Kap. 4.2). In dieser Methode vollzieht sich das eigentliche Schlüsselmanagement und die Verschlüsselung. Abb. 9 zeigt die logische Struktur eines Verschlüsselungsvorgangs. In diesem Kapitel wird dieser Vorgang detailliert erläutert.

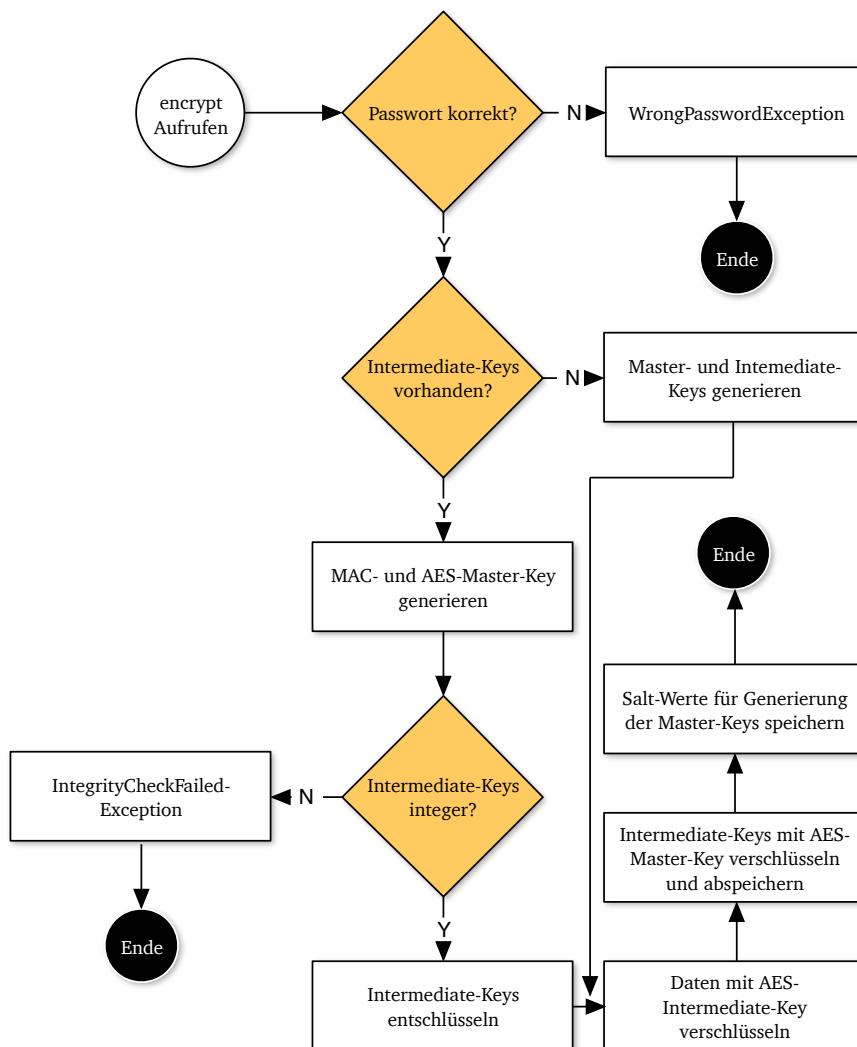


Abbildung 9: Logische Struktur Verschlüsselungsvorgang

4.3.1. Private encrypt-Methode

Die private encrypt-Methode (s. Listing 18) ist für das Schlüsselmanagement und die Verschlüsselung von Daten zuständig.

```
/*
 * Private method that handles the encryption process.
 *
 * @param integrity      The integrity mode. Choose SecureAndroid
 * .Integrity.INTEGRITY or NO_INTEGRITY.
 * @param plaintext       The plaintext as a byte array.
 * @param password        The password that will be used to derive the
 * encryption key.
 * @return                The ciphertext as a byte array.
 * @throws CryptoI0Helper.NoKeyMaterialException
 * @throws GeneralSecurityException
 * @throws CryptoI0Helper.WrongPasswordException
 * @throws CryptoI0Helper.IntegrityCheckFailedException
 * @throws CryptoI0Helper.DataNotAvailableException
 */
private byte[] encrypt(Integrity integrity, byte[] plaintext, char[]
password) throws CryptoI0Helper.NoKeyMaterialException, CryptoI0Helper
.IntegrityCheckFailedException, GeneralSecurityException,
CryptoI0Helper.WrongPasswordException, CryptoI0Helper
.DataNotAvailableException {
AESCrypto.CipherIV cipherIv;
try {
switch (integrity) {
case INTEGRITY:
// Try to load formerly stored key data and use the data to
// encrypt plaintext
SecretKeys secretKeys = getKeyData(password);
cipherIv = aesCrypto.encryptAES(plaintext, secretKeys
.getAESKey());
final byte[] mac = macCrypto.generateMAC(
cipherIv.getCipher(), secretKeys.getMACKey());
return concatenateIvAndCipherAndMAC(cipherIv, mac);
case NO_INTEGRITY:
// Try to load formerly stored key data and use the data to
// encrypt plaintext
SecretKey secretKey = getKeyDataWithoutMAC(password);
cipherIv = aesCrypto.encryptAES(plaintext, secretKey);
return concatenateIvAndCipher(cipherIv);
default:
throw new IllegalArgumentException(
WRONG_INTEGRITY_MODE_EXCEPTION);
}
}
catch (CryptoI0Helper.NoKeyMaterialException e) {
// If no key data was stored, create and store key data and use the
}
```

```

// stored key data to encrypt plaintext
switch (integrity) {
    case INTEGRITY:
        SecretKeys secretKeys = createAndStoreAndGetKeyData(
            password);
        cipherIv = aesCrypto.encryptAES(plaintext, secretKeys
            .getAESKey());
        final byte[] mac = macCrypto.generateMAC(
            cipherIv.getCipher(), secretKeys.getMACKey());
        return concatenateIvAndCipherAndMAC(cipherIv, mac);
    case NO_INTEGRITY:
        SecretKey secretKey =
            createAndStoreAndGetKeyDataWithoutMAC(password);
        cipherIv = aesCrypto.encryptAES(plaintext, secretKey);
        return concatenateIvAndCipher(cipherIv);
    default:
        throw new IllegalArgumentException(
            WRONG_INTEGRITY_MODE_EXCEPTION);
}
}
}

```

Listing 18: private encrypt-Methode

Der try-block wird durchlaufen, falls bereits Intermediate-Keys vorhanden sind. Wenn das Integrity-Flag mit dem Wert **INTEGRITY** instanziert ist, wird ein SecretKeys-Objekt mittels der Methode getKeyData initialisiert. SecretKeys enthält den AES- und den MAC-Intermediate-Key. Darauf wird der übergebene Klartext mit der Methode encryptAES aus AESCrypto verschüsselt. Auf dem resultierenden Chiffretext wird der MAC mit der Methode generateMAC aus MACCrypto berechnet. Schlussendlich wird dem Aufrufer von encrypt das Ergebnis der Methode concatenateIvAndCipherAndMAC zurückgegeben. concatenateIvAndCipherAndMAC fügt den Chiffretext, den zugehörigen IV und den MAC zu einem Byte-Array zusammen.

Wenn die Integrity-Variable den Wert **NO_INTEGRITY** annimmt, wird ein SecretKey-Objekt mit der Methode getKeyDataWithoutMAC initialisiert. SecretKey beinhaltet nur den AES-Intermediate-Key und keinen MAC-Key. Anschließend wird der Klartext mit dem AES-Schlüssel verschlüsselt und der Chiffretext mit IV durch die Methode concatenateIvAndCipher in einem Byte-Array an den Aufrufer von encrypt zurückgegeben. Sollten die Intermediate-Keys nicht vorhanden sein, lösen die Methoden getKeyData und getKeyDataWithoutMAC die NoKeyMaterialException aus. In diesem Fall springt der Programmablauf in den catch-Block. Der catch-Block ist wie der try-Block aufgebaut mit dem Unterschied, dass hier die Methode createAndStoreAndGetKeyData bzw. createAndStoreAndGetKeyDataWithoutMAC aufgerufen wird. Diese Methoden generieren die Intermediate-Keys erstmalig. getKeyData/getKeyDataWithoutMAC und createAndStoreAndGetKeyData/createAndStoreAndGetKeyDataWithoutMAC sind für das Schlüsselmanagement verantwortlich und bilden somit neben den Methoden zum Ver- und Entschlüsseln das Herzstück von SecureAndroid. Nachfolgend

werden `createAndStoreAndGetKeyData` sowie `getKeyData` detailliert erläutert. `createAndStoreAndGetKeyDataWithoutMAC` und `getKeyDataWithoutMAC` finden sich im Anhang B1.

◆ `createAndStoreAndGetKeyData`

`createAndStoreAndGetKeyData` (s. Listing 19) wird aufgerufen, wenn beim Verschlüsselungsprozess festgestellt wird, dass kein Schlüsselmaterial im App-Context (also in einer App) vorhanden ist. Dies geschieht immer, wenn eine App zum ersten Mal eine SecureAndroid-Verschlüsselungsmethode aufruft.

```
/*
 * Private method that creates the AES and MAC master-keys and stores its
 * salt values. Then the intermediate keys are generated and stored in the
 * SharedPreferences. The key then is loaded and returned to the caller.
 *
 * @param password The password from which the master key will be
 * derived.
 * @return The AES and MAC intermediate-keys used for data
 * encryption, decryption and integrity checking.
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 */
private SecretKeys createAndStoreAndGetKeyData(char[] password) throws
    GeneralSecurityException, CryptoIOHelper.DataNotAvailableException,
    CryptoIOHelper.IntegrityCheckFailedException {
    // Hash the password with salt and get the hashed password and the salt
    final PasswordCrypto.HashedPasswordAndSalt hashedPasswordAndSalt =
        passwordCrypto.hashPassword(password);
    // Generate master AES-key
    final AESCrypto.SaltAndKey aesMasterKeyAndSalt = aesCrypto
        .generateRandomAESKeyFromPasswordGetSalt(password);
    // Generate master MAC-key
    final AESCrypto.SaltAndKey macMasterKeyAndSalt = macCrypto
        .generateRandomMACKeyFromPasswordGetSalt(password);
    // Store the salt values used to generate the AES- and MAC-master-keys
    storeRootSaltData(aesMasterKeyAndSalt.getSalt(),
        macMasterKeyAndSalt.getSalt());
    // Generate the intermediate AES-key and encrypt it with the master-key
    AESCrypto.CipherIV intermediateAESCipherIV = aesCrypto.encryptAES(
        aesCrypto.generateRandomAESKey().getEncoded(),
        aesMasterKeyAndSalt.getSecretKey());
    // Generate the intermediate MAC-key and encrypt it with the master-key
    AESCrypto.CipherIV intermediateMACCipherIV = aesCrypto.encryptAES(
        macCrypto.generateRandomMACKey().getEncoded(), aesMasterKeyAndSalt
        .getSecretKey());
    // Generate MAC for encrypted AES-intermediate-key and encrypted MAC-
    // intermediate-key
    byte[] aesIntermediateMAC = macCrypto.generateMAC(
```

```

intermediateAESCipherIV.getCipher(), macMasterKeyAndSalt
.getSecretKey());
byte[] macIntermediateMAC = macCrypto.generateMAC(
intermediateMACCipherIv.getCipher(), macMasterKeyAndSalt
.getSecretKey());
// Store MACs for encrypted AES- and MAC-intermediate-keys
storeIntermediateKeysMACs(aesIntermediateMAC, macIntermediateMAC);
// Store the hashed password+salt, the intermediate-key+iv and the MAC-
// key+iv
storePasswordAndIntermediateKeyCipherIV(hashedPasswordAndSalt,
intermediateAESCipherIV, intermediateMACCipherIv);
// Load the stored keys to check if save was successful
intermediateAESCipherIV = aesCrypto.getCipherAndIVFromSharedPref(
IMEDIATE_KEY_DATA, AES_INTERMEDIATEKEY_ALIAS);
intermediateMACCipherIv = aesCrypto.getCipherAndIVFromSharedPref(
IMEDIATE_KEY_DATA, MAC_INTERMEDIATE_KEY_ALIAS);
aesIntermediateMAC = cryptoIOHelper.loadFromSharedPrefBase64(
IMEDIATE_KEY_DATA, AES_INTERMEDIATE_KEY_MAC_ALIAS);
macIntermediateMAC = cryptoIOHelper.loadFromSharedPrefBase64(
IMEDIATE_KEY_DATA, MAC_INTERMEDIATE_KEY_MAC_ALIAS);
// Check the integrity of the newly stored and then loaded intermedi-
// ate-key as failsafe mechanism
// to check if the key and mac material where generated and stored
// correctly
if (macCrypto.checkIntegrity(intermediateAESCipherIV.getCipher(),
aesIntermediateMAC, macMasterKeyAndSalt.getSecretKey()) &&
macCrypto.checkIntegrity(intermediateMACCipherIv.getCipher(),
macIntermediateMAC, macMasterKeyAndSalt.getSecretKey())) {
// Get the raw key material
final byte[] aes = aesCrypto.decryptAES(
intermediateAESCipherIV.getCipher(),
intermediateAESCipherIV.getIv(), aesMasterKeyAndSalt
.getSecretKey());
final byte[] mac = aesCrypto.decryptAES(
intermediateMACCipherIv.getCipher(),
intermediateMACCipherIv.getIv(), aesMasterKeyAndSalt
.getSecretKey());
// Generate the secret-key objects and return them
return new SecretKeys(new SecretKeySpec(aes, 0, aes.length, AES),
new SecretKeySpec(mac, 0, mac.length, MAC));
} else {
// throw exception if not
throw new CryptoIOHelper.IntegrityCheckFailedException(
INTEGRITY_CHECK_FAILED);
}
}

```

Listing 19: createAndStoreAndGetKeyData-Methode

createAndStoreAndGetKeyData berechnet zu Beginn den Hash-Wert des übergebenen Passwortes. Dazu wird die Methode hashPassword (s. Listing 20) aus PasswordCrypto aufgerufen.

```

/**
 * Hashes a password with the PBKDF2WithHmacSHA512 or the BKDF2WithHmacSHA1
 * algorithm, depending on the availability on the present platform.
 *
 * @param password The password.
 * @return The hashed password and the salt the password was hashed with.
 * @throws GeneralSecurityException
 */
protected HashedPasswordAndSalt hashPassword(char[] password) throws
    GeneralSecurityException {
    fixPrng();
    // Generate the salt
    final byte[] salt = super.generateRandomBytes(PBE_SALT_LENGTH_BYTE);
    // Instantiate key specifications with desired parameters
    final KeySpec keySpec = new PBEKeySpec(password, salt, PBE_ITERATIONS,
        KEY_LENGTH);
    // Instantiate key factory with the desired PBE-Algorithm
    final SecretKeyFactory keyFactory = SecretKeyFactory
        .getInstance(PBE_ALGORITHM);
    // Generate hash und return the hash with the salt that was used to
    // generate the hash
    return new HashedPasswordAndSalt(keyFactory.generateSecret(keySpec)
        .getEncoded(), salt);
}

```

Listing 20: hashPassword-Methode

hashPassword erstellt eine zufällige, 64 Byte lange Bitfolge und verwendet diese als Salt. Daraufhin wird ein Objekt vom Typ PBEKeySpec mit dem Passwort, dem Salt, der Anzahl an PBKDF-Iterationen und der Schlüssellänge initialisiert. Zum Berechnen des Hash-Werts wird eine SecretKeyFactory, welche vom Typ **PBE_ALGORITHM** ist, instanziert. **PBE_ALGORITHM** ist entweder mit "**PBKDF2WithHmacSHA256**" oder "**PBKDF2WithHmacSHA1**" (s. Kap. 3.2) belegt. Der Hash-Wert wird anschließend durch Aufrufen der Methode generateSecret auf dem SecretKeyFactory-Objekt berechnet. generateSecret wird die PBEKeySpec-Instanz mit den in ihr definierten Parametern übergeben. hashPassword gibt ein HashedPasswordAndSalt-Objekt zurück. Darin enthalten sind der soeben berechnete Hash-Wert und der Salt, der in die Berechnung eingeflossen ist. Der Salt muss gespeichert werden, um die Neuberechnung des Hash-Werts anhand eines übergebenen Passworts zu ermöglichen.

Nach dem Berechnen des Passwort-Hashs werden in `createAndStoreAndGetKeyData` die Master-Keys generiert. Für die Generierung des AES-Keys wird die Methode `generateRandomAESKeyFromPasswordGetSalt` der aesCrypto-Instanz aufgerufen (s. Anhang B4). Der MAC-Key wird durch Aufruf der Methode `generateRandomMACKeyFromPasswordGetSalt` (s. Anhang B2) der macCrypto-Instanz erstellt. Diese Me-

thoden haben fast den identischen Aufbau wie hashPassword. Dies liegt darin begründet, dass PBKDFs sowohl für das Hashen von Passwörtern als auch für die Generierung von zufälligen kryptographischen Schlüsseln verwendet werden können. Auch wenn PBKDFs für den Zweck des Key-Stretchings (s. Kap. 3.2) standardisiert wurden, liefern sie durch die ihr zugrunde liegenden Hash-Funktionen sehr gute Ergebnisse für das sichere Ablegen von Passwörtern. Der einzige Unterschied der Methoden zum Generieren von Schlüsseln gegenüber hashPassword besteht darin, dass das generierte Geheimnis nicht direkt, sondern in einem SecretKeySpec-Objekt zurückgegeben wird, welches wiederum zu einem SaltAndKey gehört (s. Listing 21).

```
// Generate random byte sequence for the key
final byte[] temp = keyFactory.generateSecret(keySpec).getEncoded();
// Return new key and the salt the key was created with
return new SaltAndKey(new SecretKeySpec(temp, AES_INSTANCE), salt);
```

Listing 21: Erstellen eines SecretKeys

Das SaltAndKey-Objekt enthält den Schlüssel und den Salt, mit welchem der Schlüssel erstellt wurde. Weil die Master-Keys nicht gespeichert, sondern immer wieder neu generiert werden, müssen die ihnen zugehörigen Salt-Werte abgespeichert werden. Nach Rückkehr in createAndStoreAndGetKeyData geschieht dies durch den Aufruf von storeRootSaltData. Anschließend werden der AES- und der MAC-Intermediate-Key generiert. Dies erfolgt erfolgt durch Aufruf der Methoden generateRandomAESKey und generateRandomMACKey. Der Unterschied dieser Methoden zu generateRandomAESKeyFromPasswordGetSalt/generateRandomMACKeyFromPasswordGetSalt besteht darin, dass der Schlüssel ohne einen Passwort-Input völlig pseudo-zufällig generiert wird. Eine PBKDF ist hier deswegen nicht vonnöten.

Die erstellten Intermediate-Keys werden direkt der encryptAES-Methode aus AES-Crypto übergeben und mit dem AES-Master-Key verschlüsselt. Die encryptAES-Methode (s. Listing 22) ist für das eigentliche Verschlüsseln von Daten verantwortlich.

```
/**
 * Encrypts a given plaintext with the given AES secret-key. Also generates
 * a random iv used for encrypting.
 *
 * @param plainText      The text to be encrypted.
 * @param secretKey      The AES-key to be used for encryption.
 * @return               An instance of the class CipherIV holding the
 *                      cipher and iv.
 * @throws GeneralSecurityException
 */
protected CipherIV encryptAES(byte[] plainText, SecretKey secretKey) throws
```

```

GeneralSecurityException {
    fixPrng();
    // Instantiate cipher with the AES-instance
    final Cipher cipher = Cipher.getInstance(AES_MODE);
    // Generate random bytes for the initialization vector
    final SecureRandom secureRandom = new SecureRandom();
    final byte[] initVector = new byte[IVECTOR_LENGTH_IN_BYTE];
    secureRandom.nextBytes(initVector);
    final IvParameterSpec initVectorParams = new IvParameterSpec(
        initVector);
    // Initialize cipher object with the desired parameters
    cipher.init(Cipher.ENCRYPT_MODE, secretKey, initVectorParams);
    // Encrypt
    byte[] cipherText = cipher.doFinal(plainText);
    // Return ciphertext and iv in CipherIV object
    return new CipherIV(cipherText, cipher.getIV());
}

```

Listing 22: Verschlüsselungsmethode encryptAES

encryptAES nimmt einen SecretKey und ein Byte-Array entgegen. Das Cipher-Objekt wird mit der Instanz **AES_MODE** initialisiert. **AES_MODE** ist mit dem String **"AES/CBC/PKCS5Padding"** belegt. Daraufhin wird der IV mit pseudo-zufälligen Bytes gefüllt, indem die nextBytes Methode aus SecureRandom aufgerufen wird. Der IV wird in einem IvParameterSpec-Objekt gespeichert. IvParameterSpec ist ein Container für einen IV und wird der init-Methode der Cipher-Klasse übergeben. Zusätzlich werden init das **ENCRYPT_MODE**-Flag und der Verschlüsselungsschlüssel übergeben. Dann wird doFinal aufgerufen, wodurch der Chiffretext berechnet wird. Der Chiffre-text wird gemeinsam mit dem IV in einem CipherIV-Objekt an den Aufrufer zurückgegeben. Abb. 10 zeigt das Sequenzdiagramm von encryptAES.

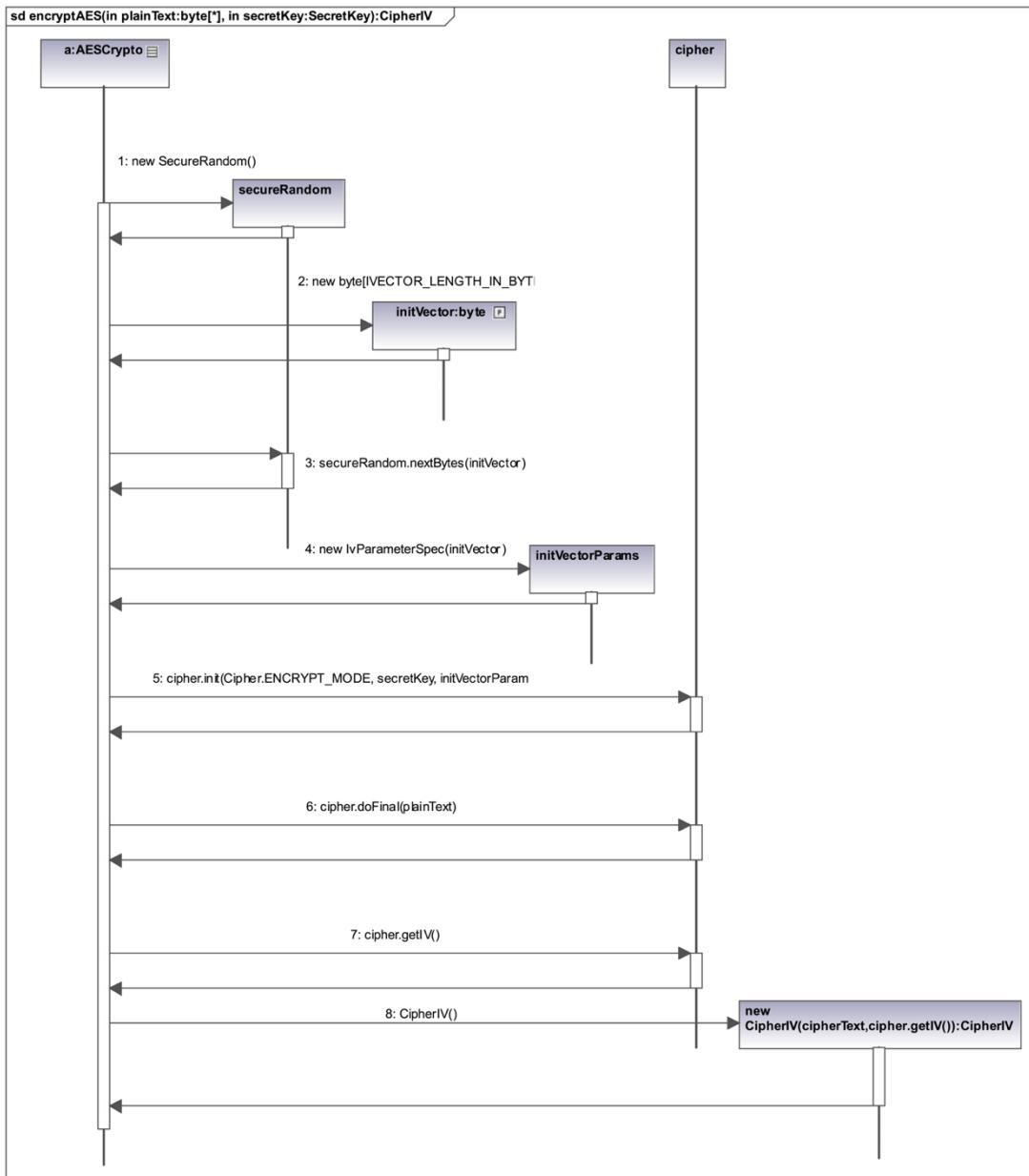


Abbildung 10: Sequenzdiagramm encryptAES

Nach Beendigung von `encryptAES` werden die MACs der Intermediate-Keys berechnet. Dazu wird die Methode `generateMAC` aus `MACCrypto` auf die Chiffretexte der Intermediate-Keys angewendet, um nach dem Prinzip von Encrypt-then-MAC vorzugehen (s. Kap. 3.3.1). Dann wird das gesamte Schlüsselmaterial in den SharedPreferences gespeichert. Das Schlüsselmaterial wird daraufhin sofort wieder geladen. Durch diesen Schritt wird sichergestellt, dass die Schlüssel nur verwendet werden, wenn sie erfolgreich gespeichert werden konnten. Ohne diesen Kontrollschritt kann es dazu kommen, dass Daten verschlüsselt werden, die dann nicht mehr entschlüsselbar sind, weil nach der Verschlüsselung ein Fehler bei der Schlüsselspeicherung auftritt.

Nachdem die MACs und die Chiffretexte mit IVs wieder erfolgreich geladen sind, wird die Integrität der Chiffretexte überprüft. Dazu wird die Methode `checkIntegrity` (s. Listing 23) aus `MACCrypto` aufgerufen.

```
/*
 * Method to check the integrity of the given ciphertext against the given
 * MAC with the given SecretKey.
 *
 * @param cipherText      The ciphertext to be checked.
 * @param mac              The MAC against which to check.
 * @param macKey           The SecretKey.
 * @return                 True if check was successful, false otherwise.
 * @throws GeneralSecurityException
 */
protected boolean checkIntegrity(byte[] cipherText, byte[] mac, SecretKey
    macKey) throws GeneralSecurityException {
    return MessageDigest.isEqual(mac, generateMAC(cipherText, macKey));
}
```

Listing 23: checkIntegrity-Methode

checkIntegrity wird der zu prüfende Chiffretext, der MAC des Chiffretextes und der MAC-Key übergeben. Die Methode überprüft, ob der mit generateMAC (s. Listing 24) generierte MAC mit dem übergebenen MAC übereinstimmt, und gibt true oder false zurück. generateMAC nimmt ein Byte-Array und einen MAC-Schlüssel entgegen und erstellt einen MAC des Byte-Arrays.

```
/*
 * Method to generate a Message Authentication Code.
 *
 * @param cipher      The ciphertext for which the MAC should be generated.
 * @param macKey     The secret key for the MAC generation.
 * @return            MAC as a byte array.
 * @throws GeneralSecurityException
 */
protected byte[] generateMAC(byte[] cipher, SecretKey macKey) throws
    GeneralSecurityException {
    final Mac mac = Mac.getInstance(MAC_ALGORITHM);
    mac.init(macKey);
    return mac.doFinal(cipher);
}
```

Listing 24: generateMAC-Methode

MAC_ALGORITHM ist entweder mit "**HmacSHA256**" oder "**HmacSHA1**" initialisiert (s. Kap. 3.2). Zum Überprüfen auf Gleichheit der MACs wird nicht die Methode `Arrays.equals` sondern `MessageDigest.isEqual` aufgerufen. Die konstante Laufzeit von `MessageDigest.isEqual` garantiert Schutz vor Timing-Angriffen auf die MAC-Verifikation (s. Kap. 3.3.2).

Ist die Integrität sichergestellt, werden die Intermediate-Keys entschlüsselt. Dies geschieht durch Aufruf der Methode `decryptAES` (s. Listing 25) aus `AESCrypto.decryptAES` erstellt aus einem Chiffretext, dem zugehörigen IV und einem AES-Schlüssel den Klartext.

```
/*
 * Decrypts a ciphertext that was encrypted with an AES-key and an
 * initialization vector.
 *
 * @param cipherText      The ciphertext to be decrypted.
 * @param iv               The initialization vector used for encryption.
 * @param secretKey        The secret key used for encryption.
 * @return                 The plaintext as a byte array.
 * @throws GeneralSecurityException
 */
protected byte[] decryptAES(byte[] cipherText, byte[] iv, SecretKey
    secretKey) throws GeneralSecurityException {
    // Instantiate cipher with the AES-instance and IvParameterSpec with
    // the iv
    final Cipher cipher = Cipher.getInstance(AES_MODE);
    final IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
    // Initialize cipher with the desired parameters
    cipher.init(Cipher.DECRYPT_MODE, secretKey, ivParameterSpec);
    // Return plaintext
    try {
        return cipher.doFinal(cipherText);
    } catch (BadPaddingException e) {
        throw new BadPaddingException(WRONG_INTEGRITY_MODE);
    }
}
```

Listing 25: Entschlüsselungsmethode `decryptAES`

`AES_MODE` ist mit dem String `"AES/CBC/PKCS5Padding"` instanziert. Das `Cipher`-Objekt wird mit dem Flag `Cipher.DECRYPT_MODE`, dem geheimen Schlüssel und dem IV initialisiert. Dann wird die Entschlüsselung vorgenommen. Mit `decryptAES` werden dann der AES- und der MAC-Intermediate-Key entschlüsselt. Beide Schlüssel werden daraufhin in einem `SecretKey`-Objekt an den Aufrufer der Methode zurückgegeben. Bei fehlgeschlagener Integritätsüberprüfung wird die `IntegrityCheckFailedException` ausgelöst.

◆ `getKeyData`

Die Methode `getKeyData` (s. Listing 26) lädt das bereits gespeicherte Schlüsselmaterial und gibt den AES- und MAC-Intermediate-Key zurück.

```

/**
 * Gets the formerly created and stored key data. Returns the AES- and MAC-
 * intermediate-keys used for encryption, decryption and integrity
 * checking.
 *
 * @param password The password from which the master key will be
 * derived.
 * @return The AES- and MAC-intermediate-keys used for data
 * encryption and decryption.
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 * @throws CryptoIOHelper.NoKeyMaterialException
 */
private SecretKeys getKeyData(char[] password) throws CryptoIOHelper
    .NoKeyMaterialException, CryptoIOHelper.IntegrityCheckFailedException,
    CryptoIOHelper.DataNotAvailableException, GeneralSecurityException,
    CryptoIOHelper.WrongPasswordException {
    // Get the formerly hashed password and its salt value from Shared
    // Preferences
    try {
        final PasswordCrypto.HashedPasswordAndSalt hashedPasswordAndSalt =
            passwordCrypto.getHashedPasswordAndSaltSharedPref(
                PASSWORD_ALIAS, PASSWORD_HASH_ALIAS, PASSWORD_SALT_ALIAS);
        // Check whether the hash of the given password+salt equals the
        // hash of the stored password
        if (passwordCrypto.checkPassword(password,
            hashedPasswordAndSalt.getHashedPassword(),
            hashedPasswordAndSalt.getSalt())) {
            // Load the salt values for generating the AES- and MAC-master-
            // keys
            final byte[] aesSalt = cryptoIOHelper
                .loadFromSharedPrefBase64(KEY_DATA_ALIAS,
                    AES_MASTERKEY_SALT_ALIAS);
            final byte[] macSalt = cryptoIOHelper
                .loadFromSharedPrefBase64(KEY_DATA_ALIAS,
                    MAC_MASTER_KEY_SALT_ALIAS);
            // Load MACs of AES- and MAC-intermediate-keys
            final byte[] intermediateAESKeyMAC = cryptoIOHelper
                .loadFromSharedPrefBase64(IMEDIATE_KEY_DATA,
                    AES_INTERMEDIATE_KEY_MAC_ALIAS);
            final byte[] intermediateMACKeyMAC = cryptoIOHelper
                .loadFromSharedPrefBase64(IMEDIATE_KEY_DATA,
                    MAC_INTERMEDIATE_KEY_MAC_ALIAS);
            // Load the encrypted AES- and MAC-intermediate-keys
            final AESCrypto.CipherIV intermediateAESCipherIV = aesCrypto
                .getCipherAndIVFromSharedPref(IMEDIATE_KEY_DATA,
                    AES_INTERMEDIATEKEY_ALIAS);
            final AESCrypto.CipherIV intermediateMACCipherIV =
                aesCrypto.getCipherAndIVFromSharedPref(IMEDIATE_KEY_DATA,

```

```

    MAC_INTERMEDIATE_KEY_ALIAS);
// Generate MAC-master-key
final SecretKey macMasterKey = macCrypto
    .generateRandomMACKeyFromPasswordSetSalt(password, macSalt);
// Check if the encrypted intermediate-keys are uncorrupted
if (macCrypto.checkIntegrity(
    intermediateAESCipherIV.getCipher(), intermediateAESKeyMAC,
    macMasterKey) && macCrypto.checkIntegrity(
    intermediateMACCipherIv.getCipher(), intermediateMACKeyMAC,
    macMasterKey)) {
    // Generate AES-master-key
    final SecretKey aesMasterKey = aesCrypto
        .generateAESKeyFromPasswordSetSalt(password,
        aesSalt);
    // Extract the raw key data for the AES- and
    // MAC-intermediate-keys
    final byte[] aes = aesCrypto.decryptAES(
        intermediateAESCipherIV.getCipher(),
        intermediateAESCipherIV.getIv(), aesMasterKey);
    final byte[] mac = aesCrypto.decryptAES(
        intermediateMACCipherIv.getCipher(),
        intermediateMACCipherIv.getIv(), aesMasterKey);
    // Return the AES- and MAC-intermediate-keys
    return new SecretKeys(new SecretKeySpec(aes, 0, aes.length,
        AES), new SecretKeySpec(mac, 0, mac.length, MAC));
} else {
    // Throw IntegrityCheckFailedException
    throw new CryptoIOHelper.IntegrityCheckFailedException(
        INTEGRITY_CHECK_FAILED);
}
} else {
    // If password check failed, throw WrongPasswordException
    throw new CryptoIOHelper.WrongPasswordException(
        WRONG_PASSWORD);
}
} catch (CryptoIOHelper.DataNotAvailableException e) {
    throw new CryptoIOHelper.NoKeyMaterialException(
        NO_KEYMATERIAL_MSG);
}
}

```

Listing 26: getKeyData-Methode

Zu Beginn wird die Methode `getHashedPasswordAndSaltSharedPref` aus `PasswordCrypto` aufgerufen, um den gespeicherten Hash-Wert des Passworts und den Salt, der zur erneuten Berechnung des Hash-Werts notwendig ist, zu laden. Die Methode liefert ein `HashedPasswordAndSalt`-Objekt mit dem Hash und dem Salt in zwei Byte-Arrays. Sollte kein gespeicherter Passwort-Hash in den SharedPreferences gefunden werden, wird die `NoKeyMaterialException` ausgelöst und `getKeyData` beendet.

Nach dem Laden des HashedPasswordAndSalt-Objekts wird geprüft, ob der Hash-Wert des an getKeyData übergebenen Passworts mit dem gespeicherten Hash-Wert übereinstimmt. Die checkPassword-Methode (s. Listing 27) aus PasswordCrypto führt diese Überprüfung aus. Dazu ruft sie hashPasswordWithSalt (s. Anhang B3) auf und vergleicht das Ergebnis mit MessageDigest.isEqual.

```
/*
 * Checks the given password and salt against the given hash value.
 *
 * @param password The password as a String.
 * @param hash      The hash as a byte array.
 * @param salt      The salt that the hash was generated with.
 * @return True if check was successful, false otherwise
 * @throws GeneralSecurityException
 */
protected boolean checkPassword(char[] password, byte[] hash, byte[] salt)
    throws GeneralSecurityException {
    return MessageDigest.isEqual(hashPasswordWithSalt(password, salt),
        hash);
}
```

Listing 27: checkPassword-Methode

Bei nicht erfolgreicher Überprüfung wird die WrongPasswordException ausgelöst und getKeyData beendet. Wenn das Passwort korrekt ist, kann das Schlüsselmaterial geladen werden. Zuerst werden die gespeicherten Salt-Werte zur Generierung der AES- und MAC-Master-Keys sowie die Chiffrentexte, IVs und MACs der Intermediate-Keys geladen. Durch Aufruf der Methode generateMACKeyFromPasswordSetSalt (s. Listing 28) wird dann der MAC-Master-Key generiert.

```
/*
 * Returns the Secret-MAC key generated with the specified salt value and
 * password.
 *
 * @param password The password used to derive the key.
 * @param salt      The salt used to derive the key.
 * @return          The secret MAC-key.
 * @throws GeneralSecurityException
 */
protected SecretKey generateRandomMACKeyFromPasswordSetSalt(char[]
    password, byte[] salt) throws GeneralSecurityException {
    fixPrng();
    final KeySpec keySpec = new PBEKeySpec(password, salt, PBE_ITERATIONS,
        MAC_128);
    final SecretKeyFactory keyFactory = SecretKeyFactory
```

```

    .getInstance(PBE_ALGORITHM);
    final byte[] temp = keyFactory.generateSecret(keySpec).getEncoded();
    return new SecretKeySpec(temp, MAC_INSTANCE);
}

```

Listing 28: generateMACKeyFromPasswordSetSalt-Methode

generateMACKeyFromPasswordSetSalt nimmt das Passwort und den Salt zur Generierung des Schlüssels entgegen. Das PBEKeySpec-Objekt wird mit dem übergebenen Passwort und Salt sowie der berechneten Iterationsanzahl für die PBKDF und der Schlüssellänge von 128 Bit initialisiert. Daraufhin wird die SecretKeyFactory mit dem PBKDF-Algorithmus instanziert. Die Methode generateSecret der keyFactory berechnet mit den übergebenen PBEKeySpec-Parametern den Schlüssel, welcher dann in einem SecretKeySpec-Objekt zurückgegeben wird. Der Unterschied von generateMACKeyFromPasswordSetSalt zu generateMACKeyFromPasswordGetSalt besteht darin, dass erstere Methode einen bereits berechneten Salt zur Generierung des Schlüssels benutzt, während die zweite Methode den Salt zufällig erstellt. Beide Methoden sind notwendig; generateMACKeyFromPasswordGetSalt erstellt einen Schlüssel erstmalig und generateMACKeyFromPasswordSetSalt rekonstruiert einen bereits erstellten Schlüssel.

Nach Generierung des MAC-Master-Keys wird in getKeyData die Integrität der geladenen Intermediate-Keys überprüft. Verläuft diese erfolgreich, wird der AES-Master-Key nach der gleichen Vorgehensweise wie der MAC-Master-Key erstellt. Mit dem AES-Master-Key werden die AES- und MAC-Intermediate-Keys entschlüsselt und in einem SecretKeys-Objekt an den Aufrufer von getKeyData zurückgegeben. Sollte die Integritätsprüfung der Intermediate-Keys fehlgeschlagen, wird die IntegrityCheckFailed-Exception ausgelöst.

Nachdem in encrypt (s. Listing 16) das Schlüsselmaterial in Form des SecretKeys-Objekts vorliegt, kann die eigentliche Verschlüsselung der übergebenen Nutzdaten stattfinden. Dazu wird die encryptAES-Methode aus AESCrypto aufgerufen und der resultierende Chiffretext mit IV in ein CipherIV-Objekt gespeichert. Wenn das Integrity-Flag mit dem Wert **INTEGRITY** instanziert ist, wird von dem Chiffretext der MAC berechnet und das Ergebnis der Methode concatenateIvAndCipherAndMAC (s. Listing 29) aus SecureAndroid an den Aufrufer von encrypt zurückgegeben. In concatenateIvAndCipherAndMAC werden der IV, der Chiffretext und der MAC des Chiffretextes in ein zusammenhängendes Byte-Array gespeichert. Dieses Byte-Array wird Base64-kodiert zurückgegeben. In SecureAndroid werden alle Daten in Base64 kodiert, um einen eventuellen Datentransfer zu vereinfachen.

```

/**
 * Private method that concatenates three byte arrays.

```

```

/*
 * @param cipherIv The cipherIv object that holds the seperate ciphertext
 * and iv arrays.
 * @param mac The MAC corresponding to the cipherIv object.
 * @return The byte array in which the iv and the ciphertext are
 * concatenated.
 */
private byte[] concatenateIvAndCipherAndMAC(AESCrypto.CipherIV cipherIv,
    byte[] mac) {
    // Create necessary arrays
    final byte[] iv = cipherIv.getIv();
    final byte[] cipher = cipherIv.getCipher();
    final byte[] ivAndCipherAndMAC = new byte[IV_LENGTH_BYTE +
        MAC_LENGTH_BYTE + cipher.length];
    // Copy the iv into the first 128 Bit of the new array
    System.arraycopy(iv, 0, ivAndCipherAndMAC, 0, IV_LENGTH_BYTE);
    // Copy the mac into the next 160/256 Bit of the new array
    System.arraycopy(mac, 0, ivAndCipherAndMAC, IV_LENGTH_BYTE,
        MAC_LENGTH_BYTE);
    // Append the cipher at the end
    System.arraycopy(cipher, 0, ivAndCipherAndMAC, MACPLUSIV_LENGTH_BYTE,
        cipher.length);
    // Return the array containing iv+mac+cipher
    return cryptoIOHelper.encodeToBase64(ivAndCipherAndMAC);
}

```

Listing 29: concatenateIvAndCipherAndMAC-Methode

Wenn das Integrity-Flag mit dem Wert **NO_INTEGRITY** instanziert ist, entfällt die Erstellung des MACs. In Szenarien wo kein MAC benötigt wird, kann durch Ausschalten der Integritätsüberprüfung die Performance um ~30 % erhöht werden. Tabelle 3 zeigt die Ergebnisse einer Performance-Messung mit und ohne Integritätsüberprüfung.

Integrität	Messung 1	Messung 2	Messung 3	Messung 4	Messung 5
INTEGRITY	543 ms	486 ms	482 ms	512 ms	506 ms
NO_INTEGRITY	332 ms	395 ms	325 ms	326 ms	333 ms
Performance-Gewinn	38.9 %	18.7 %	27 %	36.3 %	34.2%

Tabelle 3: Performance-Messungen mit und ohne Integritätsüberprüfung

4.4. Entschlüsselung

Analog zu den öffentlichen Verschlüsselungsmethoden leiten auch die öffentlichen Entschlüsselungsmethoden ihren Aufruf an die private decrypt-Methode weiter. Abb. 11 zeigt die logische Struktur eines Entschlüsselungsvorgangs. Die Entschlüsselung wird nachfolgend detailliert erläutert.

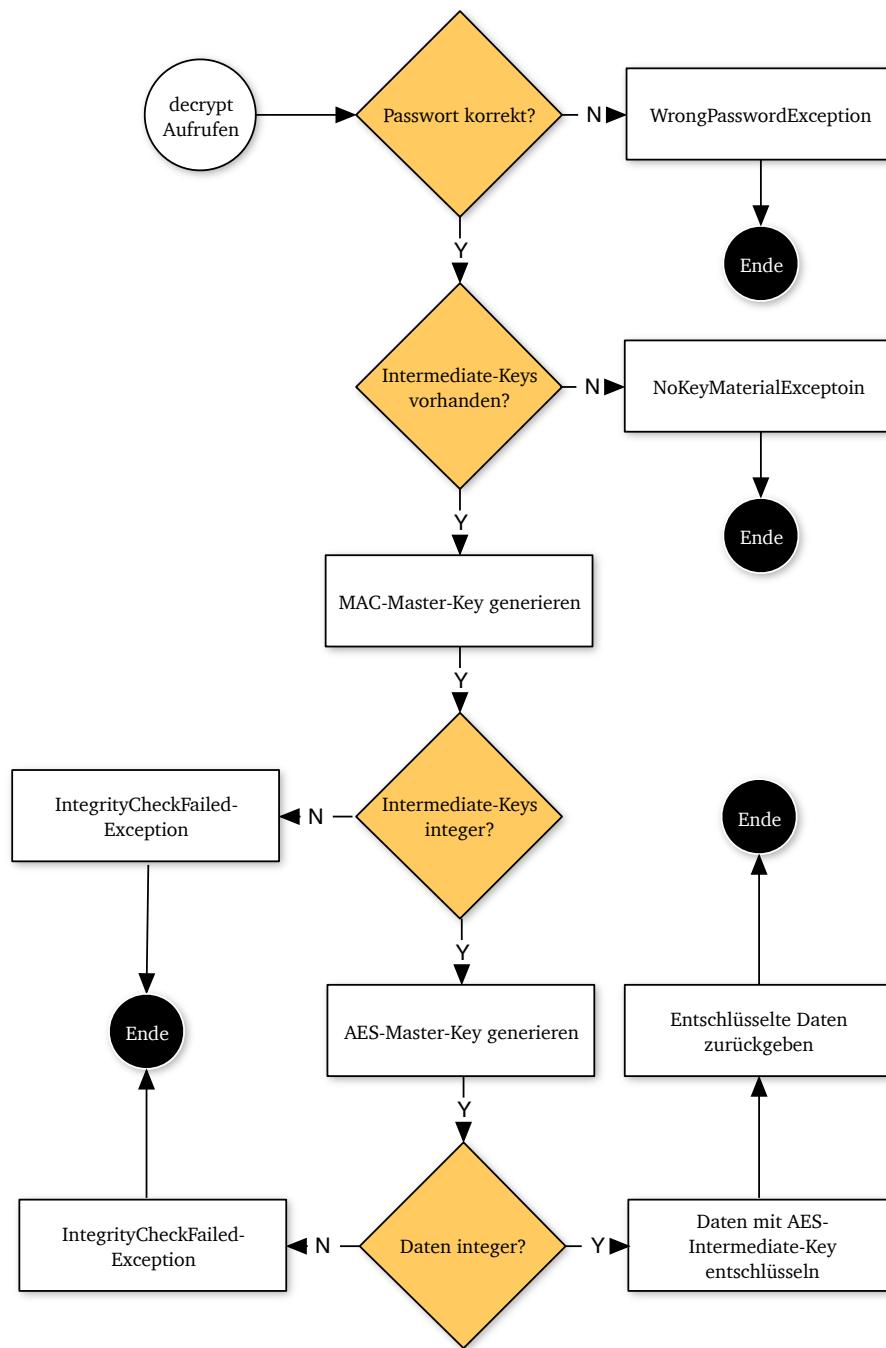


Abbildung 11: Logische Struktur Entschlüsselungsvorgang

4.4.1. Private decrypt-Methode

Die private decrypt-Methode (s. Listing 30) entschlüsselt das ihr übergebene Byte-Array und liefert den Klartext zurück.

```

/**
 * Private method that handles the decryption process.
 *
 * @param integrity      The integrity mode. Choose SecureAndroid
 * .Integrity.INTEGRITY or NO_INTEGRITY.
 * @param ivAndCipherAndMAC The ciphertext including the iv and the MAC.
 * @param password        The password that will be used to derive the
 * decryption key.
 * @return                The plaintext as a byte array.
 * @throws CryptoI0Helper.NoKeyMaterialException
 * @throws GeneralSecurityException
 * @throws CryptoI0Helper.WrongPasswordException
 * @throws CryptoI0Helper.DataNotAvailableException
 * @throws CryptoI0Helper.IntegrityCheckFailedException
 */
private byte[] decrypt(Integrity integrity, byte[] ivAndCipherAndMAC,
    char[] password) throws CryptoI0Helper.NoKeyMaterialException,
    CryptoI0Helper.IntegrityCheckFailedException, GeneralSecurityException,
    CryptoI0Helper.WrongPasswordException, CryptoI0Helper
    .DataNotAvailableException {
    // Decode the ciphertext
    final byte[] innerTempArray = cryptoI0Helper.decodeBase64(
        ivAndCipherAndMAC);
    byte[] iv = new byte[IV_LENGTH_BYTE];
    byte[] cipher;
    switch (integrity) {
        case INTEGRITY:
            final byte[] mac = new byte[MAC_LENGTH_BYTE];
            cipher = new byte[innerTempArray.length -
                (IV_LENGTH_BYTE+MAC_LENGTH_BYTE)];
            disassembleIvAndCipherAndMAC(iv, mac, cipher, innerTempArray);
            // Get the keys for integrity checking and decryption
            final SecretKeys secretKeys = getKeyData(password);
            if (macCrypto.checkIntegrity(cipher, mac, secretKeys.getMA
                CKey())) {
                // if integrity check was successful, return the decrypted
                // plaintext as byte array
                return aesCrypto.decryptAES(cipher, iv, secretKeys
                    .getAESKey());
            } else {
                throw new CryptoI0Helper.IntegrityCheckFailedException(
                    INTEGRITY_CHECK_FAILED);
            }
        case NO_INTEGRITY:
            cipher = new byte[innerTempArray.length - IV_LENGTH_BYTE];
            disassembleIvAndCipher(iv, cipher, innerTempArray);
            // Get the keys for integrity checking and decryption
            final SecretKey secretKey = getKeyDataWithoutMAC(password);
            return aesCrypto.decryptAES(cipher, iv, secretKey);
    }
}

```

```

default:
    throw new IllegalArgumentException(
        WRONG_INTEGRITY_MODE_EXCEPTION);
}
}

```

Listing 30: Die Methode decrypt aus SecureAndroid

Das der Methode gelieferte Byte-Array `ivAndCipherAndMAC` enthält den IV, den Chiffrentext und den MAC des Chiffretexts, falls bei der Verschlüsselung ein MAC erstellt wurde. Weil das Byte-Array von der `encrypt`-Methode vor Rückgabe Base64-kodiert wurde, wird dies zuerst rückgängig gemacht. Wenn in `switch` der Fall **INTEGRITY** eintritt, wird neben den Arrays für den IV und für den Ciphertext noch ein Array für den MAC angelegt. Diese Arrays werden mit den entsprechenden Längen initialisiert und der Methode `disassembleIvAndCipherAndMAC` (s. Listing 31) übergeben.

```

/*
 * Method that disassembles the concatenated CipherIvMAC-Array.
 *
 * @param iv      The initialized but empty iv array.
 * @param mac     The initialized but empty MAC array.
 * @param cipher  The initialized but empty cipher array.
 * @param ivAndCipherAndMAC The ivAndCipherAndMAC array.
 */
private void disassembleIvAndCipherAndMAC(byte[] iv, byte[] mac, byte[]
    cipher, byte[] ivAndCipherAndMAC) {
    // Copy the first 128 Bit of ivAndCipherAndMAC into iv, these contain
    // the iv
    System.arraycopy(ivAndCipherAndMAC, 0, iv, 0, IV_LENGTH_BYTE);
    // Copy Bit 129–385/289 of ivAndCipherAndMAC into mac, these contain
    // the MAC
    System.arraycopy(ivAndCipherAndMAC, IV_LENGTH_BYTE, mac, 0,
        MAC_LENGTH_BYTE);
    // Copy the remaining Bits into cipher
    System.arraycopy(ivAndCipherAndMAC, MACPLUSIV_LENGTH_BYTE, cipher, 0,
        cipher.length);
}


```

Listing 31: `disassembleIvAndCipherAndMAC`-Methode

Diese Methode kopiert den Inhalt aus dem zusammengefügten Array `ivAndCipherAndMAC` in die entsprechenden Arrays für Chiffrentext, IV und MAC.

Anschließend werden in `decrypt` die `SecretKeys` mit dem an `decrypt` übergebenen Passwort geladen. Wenn der Chiffrentext die Integritätsprüfung übersteht, wird er mit `decryptAES` entschlüsselt und an den Aufrufer von `decrypt` zurückgegeben, ansonsten wird die `IntegrityCheckFailedException` ausgelöst. Wenn der Fall **NO_INTEGRITY**

eintritt, entfällt das Laden des MAC-Keys und die Integritätsüberprüfung. Abb. 12 zeigt das Sequenzdiagramm von decryptAES.

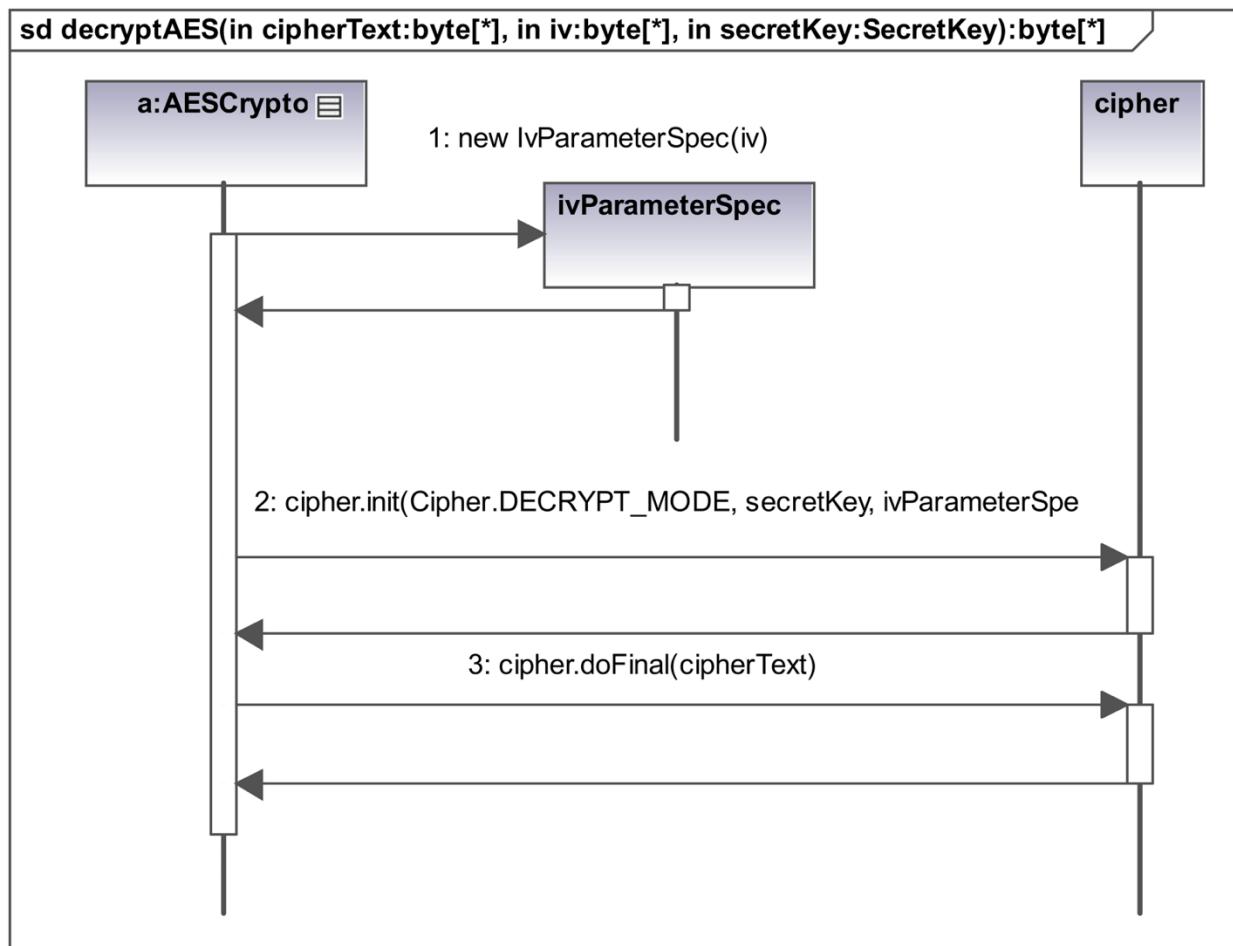


Abbildung 12: Sequenzdiagramm decryptAES

4.5. Speichern, Laden und Löschen

Die Methoden `encryptAndStore`, `encryptAndStoreWithPassword`, `retrieve` und `retrieveWithPassword` bieten neben der Verschlüsselung auch die direkte Speicherung der verschlüsselten Daten an. Diese Abstraktion von I/O-Operationen ist eine Vereinfachung für Entwickler im Umgang mit verschlüsselten Datensätzen. Des Weiteren kann mit der Methode `deleteData` ein Datensatz gezielt gelöscht werden. Die Methode `wipeKey` macht durch Löschen der Intermediate-Keys alle Daten unbrauchbar. Nachfolgend werden die Methoden, die neben dem Ver- und Entschlüsseln die Daten auch speichern und laden, detailliert behandelt.

4.5.1. Verschlüsseln und Speichern

- ◆ `encryptAndStore`

`encryptAndStore` (s. Anhang B6) prüft das Integrity-Flag und ruft daraufhin entweder die Methode `saveUserCipherMACIV` oder `saveUserCipherIV` auf. `saveUserCipherMACIV` (s. Listing 32) wird das Mode-Flag, das Ergebnis des privaten `encrypt`-Aufrufs und das Alias für die Speicherung der Daten übergeben.

```
/*
 * Private method that saves the provided array consisting of iv, cipher
 * and MAC.
 *
 * @param mode           The storage mode. Choose SecureAndroid.Mode
 * .SHARED_PREFERENCES or FILE.
 * @param ivAndCipherAndMAC The byte array containing the cipher, iv and
 * MAC.
 * @param alias          The storage alias.
 * @throws IOException
 * @throws IllegalArgumentException
 */
private void saveUserCipherMACIV(Mode mode, byte[] ivAndCipherAndMAC,
String alias) throws IOException {
    final byte[] temp = cryptoIOHelper.decodeBase64(ivAndCipherAndMAC);
    final byte[] iv = new byte[IV_LENGTH_BYTE];
    final byte[] mac = new byte[MAC_LENGTH_BYTE];
    final byte[] cipher = new byte[temp.length - (IV_LENGTH_BYTE +
        MAC_LENGTH_BYTE)];
    disassembleIVAndCipherAndMAC(iv, mac, cipher, temp);
    switch (mode) {
        case SHARED_PREFERENCES:
            aesCrypto.saveCipherAndIVToSharedPrefBase64(aesCrypto
                .instantiateCipherIV(cipher, iv), CIPHER_IV_ALIAS, alias);
            macCrypto.saveToSharedPrefBase64(MAC_USER_ALIAS, alias, mac);
            break;
        case FILE:
            aesCrypto.writeCipherAndIVToFileBase64(aesCrypto
                .instantiateCipherIV(cipher, iv), alias);
            macCrypto.saveBytesToFileBase64(alias, mac);
            break;
        default:
            throw new IllegalArgumentException(WRONG_MODE_EXCEPTION);
    }
}
```

Listing 32: `saveUserCipherMACIV`-Methode

In `saveUserCipherMACIV` wird zu Beginn das von der privaten `encrypt`-Methode gelieferte Byte-Array, welches IV, Chiffertext und MAC enthält, in eine temporäre Variable gespeichert. Dann werden Arrays für die drei zu unterscheidenden Byte-Folgen angelegt und mit der Methode `disassembleIVAndCipherAndMAC` mit den entsprechenden Werten aus dem temporären Array gefüllt.

Mit einer switch-case-Abfrage wird anschließend der Speichermodus bestimmt. Je nachdem ob die Daten in den SharedPreferences oder in einer Datei gespeichert werden sollen, werden die entsprechenden Methoden aus AESCrypto und MACCrypto aufgerufen. Beide Klassen bieten Methoden zum Speichern von Daten an. Diese Methoden greifen auf die Speichermethoden der Superklasse CryptoIOHelper zurück. Wenn das Integrity-Flag den Wert **NO_INTEGRITY** hat, wird die Methode saveUserCipherIV, die ausschließlich Ciphertext und IV speichert, aufgerufen.

◆ **encryptAndStoreWithPassword**

Die Methode `encryptAndStoreWithPassword` ist identisch zu `encryptAndStore` mit dem einzigen Unterschied, dass der privaten `encrypt`-Methode statt des automatisch generierten ein selbst gewähltes Passwort übergeben wird.

4.5.2. Laden und Entschlüsseln

◆ **retrieve/retrieveWithPassword**

Mit den öffentlichen Methoden `retrieve` und `retrieveWithPassword` werden Daten geladen, die mit `encryptAndStore` bzw. `encryptAndStoreWithPassword` verschlüsselt gespeichert wurden. Beide Methoden leiten den Aufruf an die private `retrieve`-Methode (s. Listing 33) weiter. Die private `retrieve`-Methode ist für das Laden und Entschlüsseln der angeforderten Daten zuständig.

```
/*
 * Private method that returns the data stored under the provided password
 * and alias.
 *
 * @param integrity The integrity mode. Choose SecureAndroid.Integrity
 * .INTEGRITY or NO_INTEGRITY.
 * @param mode      The storage mode. Choose SecureAndroid.Mode
 * .SHARED_PREFERENCES or FILE.
 * @param alias     The alias.
 * @param password The password.
 * @return          The decrypted data.
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.NoKeyMaterialException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws IllegalArgumentException
 * @throws IOException
 * @throws CryptoIOHelper.DataNotAvailableException
 */
private byte[] retrieve(Integrity integrity, Mode mode, String alias,
    char[] password) throws GeneralSecurityException, CryptoIOHelper
    .NoKeyMaterialException, CryptoIOHelper.IntegrityCheckFailedException,
```

```

CryptoIOHelper.WrongPasswordException, IOException,
CryptoIOHelper.DataNotAvailableException {
AESCrypto.CipherIV cipherIV;
switch (integrity) {
case INTEGRITY:
    SecretKeys secretKeys = getKeyData(password);
    cipherIV = getUserCipherIv(mode, alias);
    final byte[] mac = macCrypto.loadMAC(mode, alias);
    if (macCrypto.checkIntegrity(cipherIV.getCipher(), mac,
        secretKeys.getMACKey())) {
        return aesCrypto.decryptAES(cipherIV.getCipher(),
            cipherIV.getIv(), secretKeys.getAESKey());
    } else {
        throw new CryptoIOHelper.IntegrityCheckFailedException(
            INTEGRITY_CHECK_FAILED);
    }
case NO_INTEGRITY:
    SecretKey secretKey = getKeyDataWithoutMAC(password);
    cipherIV = getUserCipherIv(mode, alias);
    return aesCrypto.decryptAES(cipherIV.getCipher(), cipherIV
        .getIv(), secretKey);
default:
    throw new IllegalArgumentException(
        WRONG_INTEGRITY_MODE_EXCEPTION);
}
}

```

Listing 33: private retrieve-Methode

retrieve nimmt das Integrity- sowie Mode-Flag, das Daten-Alias und das Passwort entgegen. Falls das Integrity-Flag mit **INTEGRITY** instanziiert ist, wird durch Aufruf der Methode getKeyData ein SecretKeys-Objekt initialisiert. In getKeyData wird die NoKeyMaterialException ausgelöst, falls kein Schlüsselmaterial vorhanden ist. In diesem Fall wurden noch keine Daten verschlüsselt und retrieve wird beendet. retrieve kann also nur erfolgreich ausgeführt werden, wenn bereits Daten verschlüsselt und gespeichert wurden. Wenn das SecretKeys-Objekt erfolgreich initialisiert ist, wird mit der Methode getUserCipherIv (s. Listing 34) der unter dem übergebenen Alias gespeicherte Chiffretext geladen.

```

/**
 * Private method that returns a formerly stored CipherIV object.
 *
 * @param mode      The storage mode. Choose SecureAndroid.Mode
 * .SHARED_PREFERENCES or FILE.
 * @param alias     The storage alias.
 * @return          The CipherIV object.
 * @throws IllegalArgumentException
 * @throws CryptoIOHelper.DataNotAvailableException

```

```

* @throws IOException
*/
private AESCrypto.CipherIV getUserCipherIv (Mode mode, String alias) throws
    CryptoIOHelper.DataNotAvailableException, IOException {
    switch (mode) {
        case SHARED_PREFERENCES:
            return aesCrypto.getCipherAndIVFromSharedPref(CIPHER_IV_ALIAS,
                alias);
        case FILE:
            return aesCrypto.getCipherAndIVFromFile(alias);
        default:
            throw new IllegalArgumentException(WRONG_MODE_EXCEPTION);
    }
}

```

Listing 34: getUserCipherIv-Methode

getUserCipherIv überprüft den Speicherungsmodus und ruft die entsprechenden Methoden zum Laden aus den SharedPreferences oder aus dem Dateisystem auf. Weil es immer um ein CipherIV-Objekt geht, sind diese Methoden in AESCrypto implementiert. getCipherAndIVFromSharedPref (s. Listing 35) wird das Alias für die SharedPreferences-Datei und das Alias für den darin enthaltenen Datensatz übergeben. Die Methode ruft loadFromSharedPrefBase64 (s. Listing 36) aus der Superklasse CryptoIOHelper jeweils mit den übergebenen Aliasen für den Chiffrentext und den IV auf und gibt diese Daten als Byte-Arrays zurück. Mit den beiden Arrays wird ein CipherIV-Objekt instanziert und zurückgegeben.

```

/**
 * Returns a CipherIV object containing the ciphertext and initialization
 * vector stored under the specified alias.
 *
 * @param spAlias      The alias for the SharedPref.
 * @param ciphIVAlias The alias for the ciphertext and iv.
 * @return             The CipherIV instance containing the desired cipher
 * and iv.
 * @throws CryptoIOHelper.DataNotAvailableException
 */
protected CipherIV getCipherAndIVFromSharedPref(String spAlias, String
    ciphIVAlias) throws DataNotAvailableException {
    byte[] cipher = super.loadFromSharedPrefBase64(spAlias, ciphIVAlias +
        CIPHER_PART);
    byte[] iv = super.loadFromSharedPrefBase64(spAlias, ciphIVAlias +
        IV_PART);
    return new CipherIV(cipher, iv);
}

```

Listing 35: getCipherAndIVFromSharedPref-Methode

```

/**
 * Reads the data saved under the specified alias. Data must have been
 * saved Base64-encoded.
 *
 * @param spAlias      The alias under which the SharedPref was stored.
 * @param dataAlias    The alias of the data itself.
 * @return             The data as a byte array.
 * @throws CryptoI0Helper.DataNotAvailableException
 */
protected byte[] loadFromSharedPrefBase64(String spAlias, String dataAlias)
    throws DataNotAvailableException {
    final SharedPreferences sharedPreferences = context
        .getSharedPreferences(spAlias, Context.MODE_PRIVATE);
    String temp = sharedPreferences.getString(dataAlias, null);
    if (temp != null) {
        return decodeBase64String(temp);
    } else {
        throw new DataNotAvailableException(DATA_NOT_AVAILABLE);
    }
}

```

Listing 36: loadFromSharedPrefBase64-Methode

Die Methode `loadFromSharedPrefBase64` wird mit zwei Aliasen aufgerufen. `spAlias` bezeichnet die `SharedPreferences`-XML-Datei. `dataAlias` bezeichnet den Schlüssel des gesuchten Wertes innerhalb der Datei. Die Datei wird im privaten Modus geöffnet und mit `getString` wird der geladene Wert in eine temporäre String-Variablen gespeichert. Sollte der Wert unter dem angegebenen Schlüssel nicht existieren, liefert `getString` `null` zurück. In diesem Fall wird die `DataNotAvailableException` ausgelöst. Ist der Wert vorhanden, wird er dekodiert und an `getCipherAndIVFromSharedPref` zurückgegeben.

Wenn `getUserCipherIV` durch das Flag `Mode.FILE` feststellt, dass eine im Dateisystem abgespeicherte Datei geladen werden soll, wird die Methode `getCipherAndIVFromFile` (s. Listing 37) aufgerufen.

```

/**
 * Returns a CipherIV object containing the ciphertext and initialization
 * vector stored under the specified filename.
 *
 * @param filename  The filename.
 * @return          The CipherIV instance containing the desired cipher and
 *                 iv.
 * @throws IOException
 */

```

```
/*
protected CipherIV getCipherAndIVFromFile(String filename) throws
IOException {
    byte[] cipher = super.readBytesFromFile(filename + CIPHER_PART);
    byte[] iv = super.readBytesFromFile(filename + IV_PART);
    return new CipherIV(cipher, iv);
}
```

Listing 37: getCipherAndIVFromFile-Methode

`getCipherAndIVFromFile` lädt Daten, die in einer Datei abgespeichert wurden. Auch hier wird die Methode `readBytesFromFile` (s. Listing 38) jeweils mit den Parametern für den Chiffertext und den IV aufgerufen. Die erhaltenen Byte-Arrays werden dann als `CipherIV`-Objekt zurückgegeben.

```
/**
 * Reads a formerly saved byte array from the given file in the app folder,
 * must have been Base64-encoded.
 *
 * @param filename      The filename.
 * @return              The data as a byte array.
 * @throws IOException
 */
protected byte[] readBytesFromFile(String filename) throws IOException {
    final FileInputStream fis = context.openFileInput(filename);
    final byte[] buffer = new byte[(int) fis.getChannel().size()];
    fis.read(buffer);
    fis.close();
    return this.decodeBase64(buffer);
}
```

Listing 38: readBytesFromFile-Methode

`readBytesFromFile` öffnet einen `FileInputStream` durch Aufrufen der Methode `openFileInput` der `Context`-Variable. Daraufhin wird ermittelt, wie groß die Datei ist und ein Byte-Array mit der entsprechenden Größe angelegt. Dann wird der Dateiinhalt in das Array kopiert welches von `readBytesFromFile` Base64-dekodiert und zurückgegeben wird.

In `retrieve` ist das `CipherIV`-Objekt nach der erfolgreichen Ausführung von `getUserCipherIV` initialisiert. Daraufhin muss der dem Chiffertext zugehörige MAC geladen werden. Dazu wird die Methode `loadMAC` aus `MACCrypto` aufgerufen. `loadMAC` ruft – analog zu den soeben besprochenen Methoden aus `AESCrypto` – die Lademethoden der Superklasse `CryptoIOHelper` auf. Nach dem Laden des MACs wird der in `CipherIV` enthaltene Chiffertext auf Integrität überprüft und bei bestandener Prüfung entschlüsselt an den Aufrufer zurückgegeben. Bei nicht erfolgreicher Integritätsprüfung wird die

`IntegrityCheckFailedException` ausgelöst. Falls das Integrity-Flag den Wert `NO_INTEGRITY` hat, entfallen das Laden des MACs und die Integritätsüberprüfung. In diesem Fall wird der Chiffertext direkt entschlüsselt und an den Aufrufer zurückgegeben.

4.5.3. Gezieltes Löschen

Die Methode `deleteData` (s. Kap. 4.2) wird dazu verwendet, verschlüsselte gespeicherte Daten gezielt zu löschen. `deleteData` nimmt das Mode-Flag und das Daten-Alias entgegen. Durch die Überprüfung des Speichermodus wird entschieden, ob die Methode zum Löschen aus den SharedPreferences oder die Methode zum Löschen einer Datei aufgerufen wird. `deleteCipherAndIVFromSharedPref` bzw. `deleteCipherAndIVFile` (s. Anhang B7) löscht die unter dem übergebenen Alias gespeicherten Daten. Dazu werden die Methoden `deleteFromSharedPref` und `deleteFile` (s. Anhang B8) aus `CryptoIOHelper` mit den entsprechenden Aliasen für die zu löschen Daten aufgerufen.

4.5.4. Alle Daten löschen

Um alle mit SecureAndroid verschlüsselten und gespeicherten Daten zu löschen, wird die `wipeKey`-Methode, die in Kapitel 4.2 vorgestellt wurde, verwendet. Die Methode löscht alle unter den Aliasen `IMEDIATE_KEY_DATA`, `KEY_DATA_ALIAS` und `PASSWORD_ALIAS` gespeicherten Daten. Sie ruft dazu `deleteSharedPref` (s. Anhang B8) aus `CryptoIOHelper` auf. Durch das Löschen des Schlüsselmaterials können die verschlüsselten Daten nicht mehr entschlüsselt werden. Diese Vorgehensweise ist möglich, weil die Intermediate-Keys zufällig erstellt werden. Der erstellte Schlüssel kann somit nicht rekonstruiert werden. Die einzige Möglichkeit zu versuchen, die Daten zu entschlüsseln, ist ein Brute-Force-Angriff auf den Schlüssel. Mit diesem Angriff ist der Schlüssel in polynomialer Zeit aber nicht zu berechnen und die Daten sind somit unbrauchbar. (s. Kap. 3.2).

4.6. Testen

Bei der Softwareentwicklung ist das Testen des geschriebenen Programmcodes eine der wichtigsten Aufgaben, um die Qualität des Produkts zu sichern [24]. Zu diesem Zweck wird SecureAndroid auf drei Arten getestet:

- ◆ Unit-Tests
- ◆ Beispiel-App
- ◆ Testzweig eap-Apps

4.6.1. Unit-Tests

Unit-Tests verifizieren, dass Software-Komponenten die an sie gestellten funktionalen Anforderungen erwartungsgemäß erfüllen. Die Tests werden auf Programmcode-Ebene durchgeführt. Das bedeutet, dass *Methoden* und deren Zusammenwirken getestet werden. Das Android-SDK von Google bietet verschiedene Testmöglichkeiten, darunter auch das Testen mit JUnit. Für das Testen von SecureAndroid wurden drei JUnit-Testklassen angelegt:

◆ LogicalTests

In der Klasse `LogicalTests` wird getestet, ob sich die Ver- und Entschlüsselungsmethoden logisch korrekt verhalten. Exemplarisch dafür ist die Methode `testArrayProvidedPasswordEncryptionDecryption` (s. Listing 39). Diese Methode überprüft, ob ein String nach dem Ver- und Entschlüsseln identisch zu dem String vor dem Ver- und Entschlüsseln ist.

```
/*
 * Tests if dec(enc(string)) == string with the provided password
 * @throws Exception
 */
@Test
public void testArrayProvidedPasswordEncryptionDecryption() throws
    Exception {
    // Wipe all key material
    secureAndroid.wipeKey();
    final byte[] ciphertext = secureAndroid.encryptWithPassword(
        SecureAndroid.Integrity.INTEGRITY, testStringOne.getBytes("UTF-8"),
        testPasswordOne);
    final byte[] plainbytes = secureAndroid.decryptWithPassword(
        SecureAndroid.Integrity.INTEGRITY, ciphertext, testPasswordOne);
    assertEquals(testStringOne, new String(plainbytes, "UTF-8"));
}
```

Listing 39: `testArrayProvidedPasswordEncryptionDecryption`-Methode

Zu Beginn der Methode wird jegliches Schlüsselmaterial gelöscht, weil jede Testmethode in sich selbst logisch geschlossen ist. Dann wird ein Test-String mit einem Testpasswort verschlüsselt und der Chiffertext in das Byte-Array `cipherText` gespeichert. Der Chiffertext wird wieder entschlüsselt, und der entschlüsselte Text wird mit dem ursprünglichen Test-String auf Gleichheit verglichen. Bei Gleichheit ist der Test erfolgreich. Abb. 13 zeigt das UML-Klassendiagramm von `LogicalTests`.



Abbildung 13: UML-Klassendiagramm LogicalTests

◆ OtherExceptionTests

Mit der Testklasse `OtherExceptionTests` werden die in `CryptoI0Helper` definierten Exceptions `IntegrityCheckFailedException`, `NoKeyMaterialAvailableException`, `WrongModeException` und `DataNotAvailableException` getestet. Beispielhaft wird nachfolgend die Methode zum Testen der `IntegrityCheckFailedException` (s. Listing 40) erläutert.

```

/**
 * Tests whether the integrity check for the MAC detects a manipulation.
 * @throws Exception
 */
@Test
public void integrityCheckFailedExceptionTest() throws Exception {
  secureAndroid.wipeKey();
  final CryptoI0Helper cryptoI0Helper = new CryptoI0Helper(getContext());
  byte[] cipherText = secureAndroid.encrypt(SecureAndroid.Integrity

```

```

    .INTEGRITY, testStringOne.getBytes("UTF-8"));
cipherText = cryptoIOHelper.decodeBase64(cipherText);
cipherText[1] = (byte) new Random().nextInt();
cipherText[20] = (byte) new Random().nextInt();
exception.expect(CryptoIOHelper.IntegrityCheckFailedException.class);
secureAndroid.decrypt(SecureAndroid.Integrity.INTEGRITY,
    cryptoIOHelper.encodeToBase64(cipherText));
}

```

Listing 40: integrityCheckFailedExceptionTest-Methode

In `integrityCheckFailedExceptionTest` wird ein Test-String verschlüsselt und der Chiffretext in das Byte-Array `cipherText` gespeichert. Der Chiffretext wird mit zwei zufällig generierten Werten an zwei Stellen manipuliert. Dies simuliert die Manipulation eines Angreifers. Das manipulierte Array wird dann der `decrypt`-Methode übergeben. Weil der manipulierte Chiffretext unterschiedlich von dem Chiffretext ist, auf dem der MAC berechnet wurde, wird nun erwartet, dass auch der auf dem manipulierten Chiffretext berechnete MAC unterschiedlich von dem ursprünglich berechneten MAC ist. SecureAndroid sollte diesen Umstand bemerken und die `IntegrityCheckFailedException` auslösen. In diesem Fall ist der Test bestanden.



Abbildung 14: UML-Klassendiagramm OtherExceptionTests

◆ PasswordTests

PasswordTests prüft, ob die WrongPasswordException korrekt ausgelöst wird. Des Weiteren werden die changePassword-Methoden überprüft. Die WrongPasswordException tritt immer dann auf, wenn Daten mit einem anderen als mit dem im Applications-Context geltenden Passwort ver- oder entschlüsselt werden. Es kann im Context immer nur ein Passwort geben. Der offensichtlichste Fall, der zur Auslösung der WrongPasswordException führen sollte, wird mit der Methode passwordTestFileProvidedPasswordEncryptionDecryption (s. Listing 41) simuliert. Dies ist der Fall, dass für das Entschlüsseln ein anderes Passwort als für das Verschlüsseln verwendet wird.

```
/*
 * Tests if WrongPasswordException is thrown when wrong password is
 * provided
 * @throws Exception
 */
@Test
public void passwordTestFileProvidedPasswordEncryptionDecryption() throws
    Exception {
    // Wipe all key material
    secureAndroid.wipeKey();
    secureAndroid.encryptAndStoreWithPassword(SecureAndroid.Integrity
        .INTEGRITY, SecureAndroid.Mode.FILE, testStringOne.getBytes(
        "UTF-8"), testAliasOne, testPasswordOne);
    exception.expect(CryptoIOHelper.WrongPasswordException.class);
    secureAndroid.retrieveWithPassword(SecureAndroid.Integrity.INTEGRITY,
        SecureAndroid.Mode.FILE, testAliasOne, testPasswordTwo);
}
```

Listing 41: passwordTestFileProvidedPasswordEncryptionDecryption-Methode

Zum Verschlüsseln wird **testPasswordOne** und zum Entschlüsseln **testPasswordTwo** verwendet. Weil **testPasswordTwo** verschieden von **testPasswordOne** ist, wird erwartet, dass SecureAndroid die WrongPasswordException auslöst. In diesem Fall ist der Test bestanden.

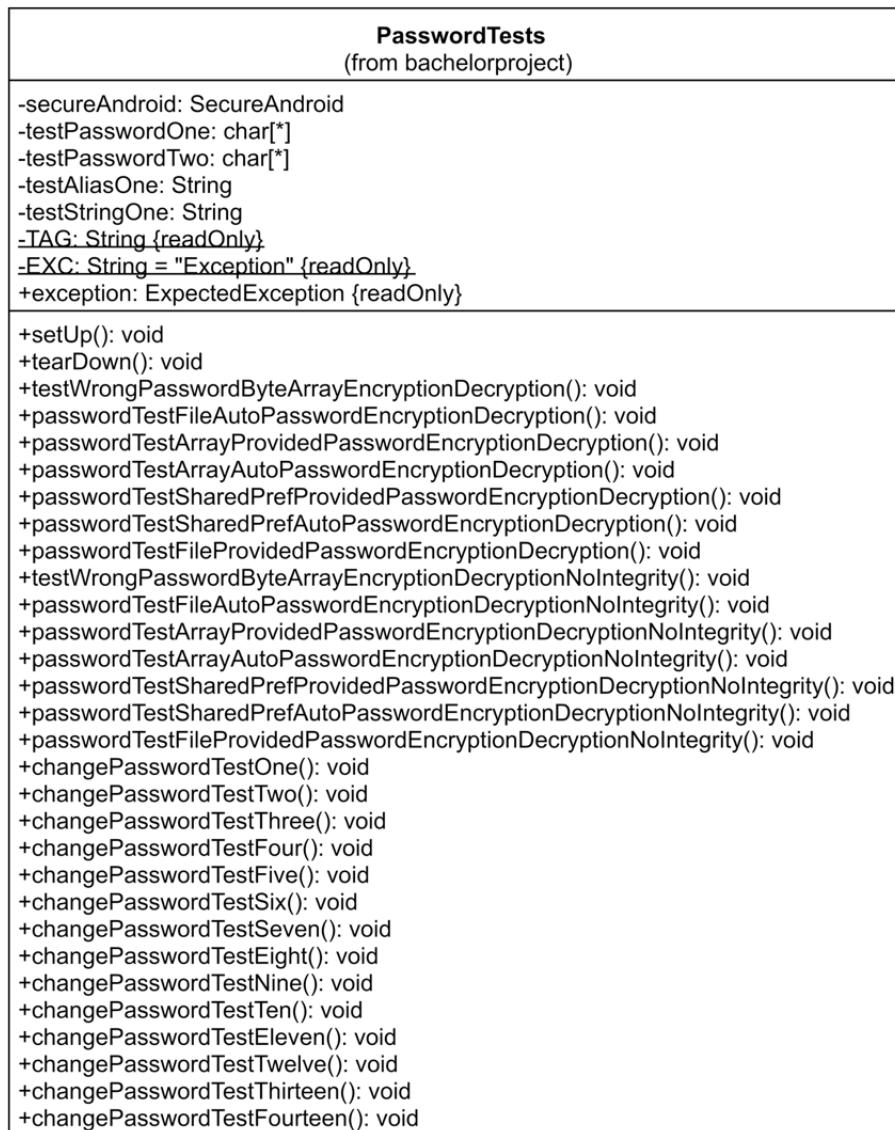


Abbildung 15: UML-Klassendiagramm PasswordTests

Insgesamt enthalten die Testklassen 77 Unit-Tests. Die Tests wurden auf folgenden Geräten ausgeführt:

- Lenovo A5500-H (Android 4.4.2)
- Samsung Galaxy Ace 2 (Android 4.1.2)
- Motorola Moto G (Android 5.1)
- Google Nexus 5X virtuelle Maschine (Android 6.0)

Auf allen Geräten erreichen alle Tests den Status *grün*. Die Tests verlaufen somit erfolgreich und stellen sicher, dass SecureAndroid sich wie erwartet verhält.

4.6.2. Testen mit einer Beispiel-App

Während Unit-Tests eine Software auf Programmcode-Ebene testen, muss Software auch unter realen Bedingungen getestet werden [24]. Zu diesem Zweck wird im Rahmen dieser Arbeit eine Beispiel-App entwickelt. Diese App vereint alle Funktionalitäten von SecureAndroid zum Ver- und Entschlüsseln in einem User-Interface (UI). Die App bietet die Möglichkeit an, einen String zu verschlüsseln und dafür einen Verschlüsselungsmodus auszuwählen. Danach kann der String wieder entschlüsselt werden. Durch die verfügbaren Modi werden alle von SecureAndroid zur Verfügung gestellten Verschlüsselungsmöglichkeiten abgedeckt. Nach Verschlüsselung wird der Chiffretext in eine Variable gespeichert und in der App angezeigt. Dies ist in Abb. 16 ersichtlich.

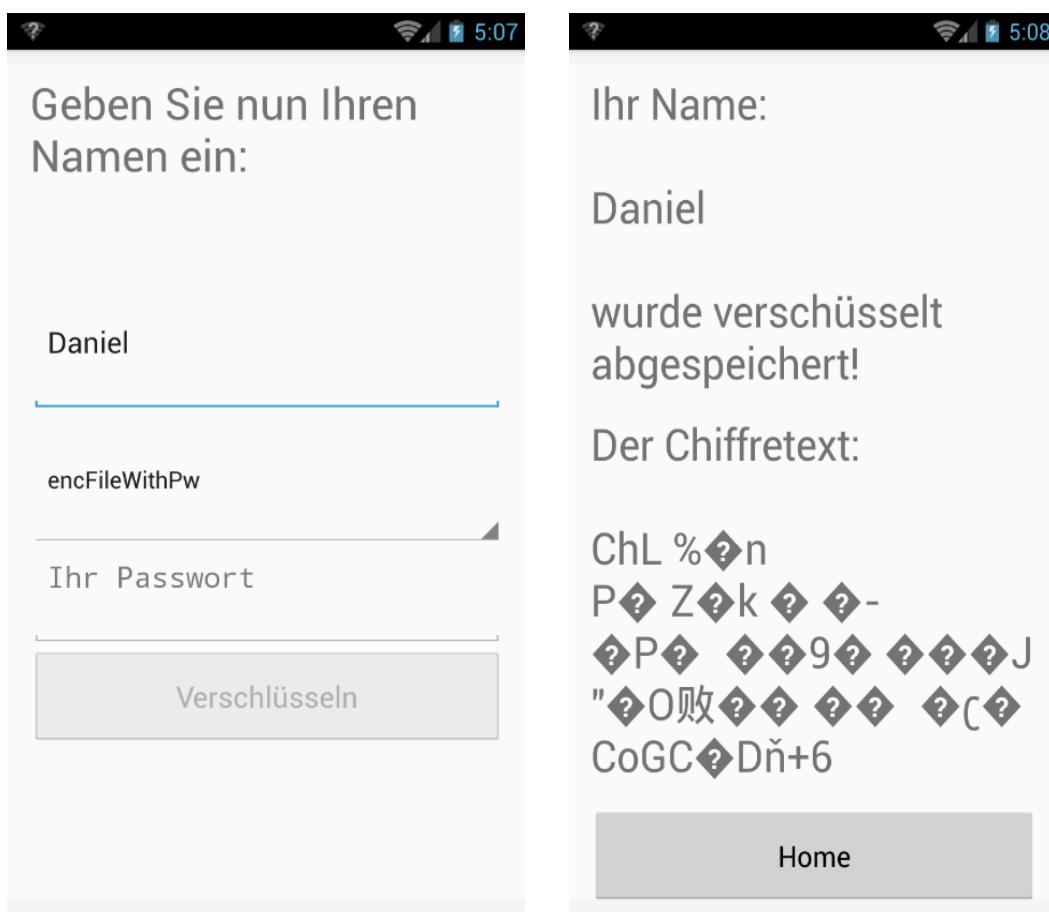


Abbildung 16: Verschlüsselung Beispiel-App

Als nächstes wird überprüft, ob die Daten korrekt verschlüsselt als Datei oder in den SharedPreferences gespeichert werden, wenn `encryptAndStore/encryptAndStoreWithPassword` aufgerufen werden. Mit dem *Android-Device-Monitor* aus der Entwicklungsumgebung *Android-Studio*⁴ von Google kann auf das Android-Dateisystem zugegriffen werden. In Abb. 17 ist ein Screenshot des Device-Monitors zu sehen, nachdem

⁴ <http://developer.android.com/tools/studio/index.html>

mit SecureAndroid ein String sowohl in einer Datei als auch in den SharedPreferences gespeichert wurde. Die von SecureAndroid erstellten Dateien sind markiert.

Android Device Monitor					
	File Explorer		System Information		
Name	Size	Date	Time	Permissions	Info
com.android.wallpaper		2015-09-30	15:37	drwxr-x--x	
com.android.wallpaper.holospiral		2015-09-30	15:37	drwxr-x--x	
com.android.wallpaper.livepicker		2015-09-30	15:37	drwxr-x--x	
com.conenergy.android		2016-02-09	12:49	drwxr-x--x	
com.conenergy.android.beta		2016-02-09	12:49	drwxr-x--x	
com.conenergy.android.schwerte_app		2016-02-09	12:49	drwxr-x--x	
com.conenergy.android.schwerte_app.beta		2016-02-09	12:49	drwxr-x--x	
com.cyanogenmod.filemanager		2015-09-30	15:37	drwxr-x--x	
com.example.android.apis		2016-02-09	12:49	drwxr-x--x	
com.example.android.livecubes		2015-09-30	15:37	drwxr-x--x	
com.genymotion.superuser		2015-09-30	15:37	drwxr-x--x	
com.svox.pico		2015-09-30	15:37	drwxr-x--x	
de.ewr_gmbh.energiebuendel.beta		2016-02-09	12:49	drwxr-x--x	
de.mvv.meinquadrat.beta		2016-02-09	12:49	drwxr-x--x	
dsahm.bachelorproject		2016-02-09	12:50	drwxr-x--x	
app_dxmaker_cache		2015-12-17	13:52	drwxrwx--	
cache		2015-12-17	13:26	drwxrwx--x	
files		2016-02-09	12:53	drwxrwx--x	
FILE_ALIAS	44	2016-02-09	12:51	-rw-rw----	
FILE_ALIASCipher	24	2016-02-09	12:51	-rw-rw----	
FILE_ALIASiv	24	2016-02-09	12:51	-rw-rw----	
lib		2016-02-09	12:50	lrwxrwxrwx -> /data/a...	
shared_prefs		2016-02-09	12:51	drwxrwx--x	
SecureAndroid.CipherV.alias.xml	194	2016-02-09	12:51	-rw-rw----	
SecureAndroid.Intermediate.Key.Data.xml	664	2016-02-09	12:51	-rw-rw----	
SecureAndroid.IterationCount.alias.xml	138	2015-12-17	13:26	-rw-rw----	
SecureAndroid.Key.Data.alias.xml	360	2016-02-09	12:51	-rw-rw----	
SecureAndroid.MACIV.Alias.xml	312	2016-02-09	12:51	-rw-rw----	
SecureAndroid.Password.Alias.xml	304	2016-02-09	12:51	-rw-rw----	
dsahm.bachelorproject.test		2016-02-09	12:49	drwxr-x--x	
jp.co.omronsoft.openwnn		2015-09-30	15:37	drwxr-x--x	
don'tpanic		2015-09-30	15:36	drwxr-x--x	
dirm		2015-09-30	15:37	drwxrwxr--	
local		2015-09-30	15:36	drwxrwx--x	
lost+found		1970-01-01	00:00	drwxrwx---	
media		2015-09-30	15:36	drwxrwx---	
misc		2015-09-30	15:36	drwxrwx--t	
property		2016-02-09	12:49	drwx----	

Abbildung 17: Android-Device-Monitor

Nach dem Verschlüsseln und Speichern in einer Datei ist die entsprechende Datei *FILE_ALIASCipher* (s. Abb. 17) mit folgendem Inhalt gefüllt:

/8pDJivL9mmCYAXjGVThBg==

Dies ist der Base64-kodierte Chiffretext. Nach Dekodierung kommt der eigentliche Chiffretext zu Tage:

ÿÊC&+Ëöi, `äTá

Die Speicherung der Daten in den SharedPreferences führt zu dem gleichen Ergebnis. Die Verschlüsselung und Speicherung ist somit nachweislich erfolgreich verlaufen. Als letztes muss die Entschlüsselung getestet werden. Die App bietet die Möglichkeit, den String in dem Modus, in welchem er verschlüsselt wurde, wieder zu entschlüsseln. Dies ist in Abb. 18 ersichtlich.

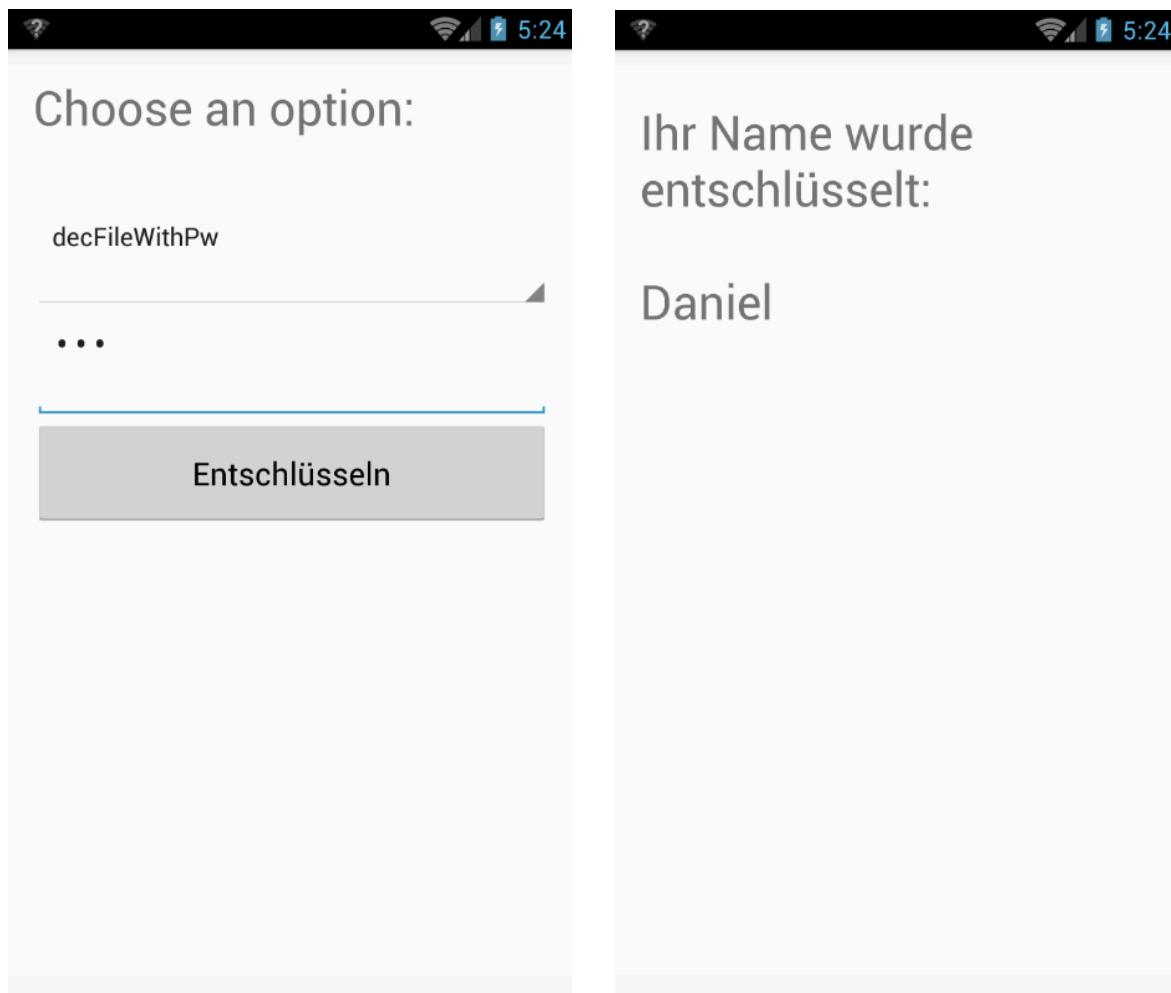


Abbildung 18: Entschlüsselung Beispiel-App

4.6.3. Testen mit der eap-App Stadtwerkzeug

eap bietet White-Label-Apps für Stadtwerke an. Das Grundgerüst für die Apps von verschiedenen Stadtwerken ist identisch. Diese Hauptfunktionalitäten sind im core-package implementiert. Die konkreten Ausprägungen für verschiedene Stadtwerke erben von core und werden individualisiert. Als Test-App für SecureAndroid dienen alle Apps von eap (s. Kap. 6.1). Exemplarisch soll der Test mit der App der Stadtwerke Düsseldorf, dem *Stadtwerkzeug (swd)*⁵, vorgestellt werden. Die App besitzt drei Funktionalitäten, die das Speichern und Laden von personenbezogenen Daten zur Folge haben: Das Speichern von Zählernummern- und ständen inkl. Vertragskontonummer, einen Abfallkalender, welcher anzeigt, wann an der angegebenen postalischen Adresse die Müllabfuhr die verschiedenen Tonnen abholt, und die Möglichkeit, einen Energie-Verbrauchsvergleich mit Wohnungsparametern durchzuführen. Die Daten werden als Datei im JSON-Format⁶ im Dateisystem von Android gespeichert. Die Speicherung von Zählerständen und Nutzerdaten erfolgt dabei jeweils in einer eigenen Datei.

⁵ <http://www.stadtwerkzeug.de/>

⁶ <http://www.json.org/>



Abbildung 19: Screenshots swd-App

◆ Speicherung Zähler und Zählerstände

Die Speicherung der Zählerstände findet in `core` statt. Sie muss dementsprechend nur einmal für alle Apps angepasst werden. Die zu verändernden Methoden sind die Methoden `save` und `load` in der Klasse `MeterReader` (s. Anhang B10). Listing 42 zeigt die um `SecureAndroid` erweiterte `save`-Methode.

```
public static boolean save(final List<EnergyMeter> values, final Context context) {
    try {
        final Gson gson = new Gson();
        final String jsonRep = gson.toJson(values);
        final SecureAndroid secureAndroid = new SecureAndroid(context, 1000);
        // For encryption
        secureAndroid.encryptAndStore(SecureAndroid.NO_INTEGRITY,
            SecureAndroid.SHARED_PREFERENCES, jsonRep
                .getBytes("UTF-8"), "meters");
        // For testing
        cryptoIOHelper.saveStringToSharedPref("SecureAndroid
            .CipherIV.alias", "meters_plain", jsonRep);
        return true;
    } catch (NoSuchAlgorithmException|InvalidKeySpecException
        |UnsupportedEncodingException ex) {
        return false;
    } catch (CryptoIOHelper.NoAlgorithmAvailableException
```

```

|CryptoI0Helper.NoKeyMaterialException|CryptoI0Helper
.IntegrityCheckFailedException|IOException|CryptoI0Helper
.WrongModeException|CryptoI0Helper.WrongPasswordException|
CryptoI0Helper.DataNotAvailableException|
GeneralSecurityException e) {
return false;
}
}

```

Listing 42: save-Methode

save nimmt die zu speichernden EnergyMeter (EnergyMeter repräsentiert einen Zähler) als Liste entgegen. Ein EnergyMeter-Objekt enthält wiederum eine Liste mit Zählerständen. Weil die Verschlüsselungsmethoden aus SecureAndroid ein Byte-Array erwarten, wird eine Byte-Repräsentation der EnergyMeter-Liste benötigt. EnergyMeter implementiert nicht das Interface Serializable, deshalb kann die Liste nicht serialisiert werden. Aus diesem Grund wird die von Google bereitgestellte Klasse Gson verwendet, um das Objekt in seine JSON-Repräsentation zu transformieren. Die Methode toJson liefert einen String im JSON-Format. Dieser String enthält alle Informationen der EnergyMeter-Liste. Aus diesem String kann mit getBytes das benötigte Byte-Array erzeugt werden, welches mit encryptAndStore verschlüsselt und unter dem Alias **"meterscipher"** in den SharedPreferences gespeichert wird. Zu Testzwecken wird als Vergleichswert der JSON-String unverschlüsselt unter dem Alias **"meters_plain"** in der gleichen SharedPreferences-Datei gespeichert. Abb. 20 zeigt einen Screenshot des Inhaltes der XML-Datei. Es ist zu erkennen, dass die Daten unter dem Alias **"meterscipher"** korrekterweise verschlüsselt und unter dem Alias **"meters_plain"** unverschlüsselt abgespeichert wurden. Der Verschlüsselungsvorgang ist folglich erfolgreich. Um die save-Methode in die Produktiv-Umgebung zu überführen, muss die Speicherung des Klartexts entfernt werden.

```

<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="meterscipher">ÍszÈÜENQÔÙ(FF11.Łky [SUBUP]Ù:ŁoŁ[CAN]H>SI BS6>yôÝSOHDC4z(
SI BELaÑÜSOH 'DC4 ?e@cy2þ"SDC2 bæº,»TEMJ]SOH GJ«[06 7*iô$SI àØLø(J;jqJ.ØWQôSYN cåÅ0%ØÖý>
oSYN UENQWEn>1ENQ^EMDyIA)AENO 'áotk-
ÅKRSU«ETB iâé;UôåäETB ØSUB-706S07Ù'a'NÅðeí4 ñ2?UØ27Ù#P=DSFF ædSUB Ø6SI 1ETX "z^ÓßíjBåÈ±D§*|è#à
â#i+3@%«[ENQ YíÈWöØRS !½KtI²+mþ¹{DLE eðsiKÅYØBEL )ÅUÉi8C DLE çåôSO Ia:DC4 ÈÉA
ÐVK:@/ _»4"È>EBa!éOLN[BS PS:9ENQ ÜBS YpØ?zVTØ}DLE üÜG £ aüJ1ETX iÅpCKRS EBU" DLE G%ñ8
;FéØéÈ§dNN0s~¾xiâéY74¥_öngiDLE mužåé ³ Öré7X
·ETB zE(½1å±iùACKY-"æFDC3 @~SI DC3 @ US Z8\BS Ü0j:nKwtç!Bf!DC3~2ØKiJ<-ACK çÓÑæwd
fRi%?tv=FS qw#_ÜÜCANeg[éØÅ@s(02~SUS 2Ø%FFÄÑI½æBEL ;FS dÅCôSUB /VT åxÜIX6Ù;US ~Ís)ü8FFWDöög
C{ETB CAN BEL @Œ, ØY CAN WØ n³ééþ+ XëÅÆEW YØlðox<øOT>FS æØØFBT;ETBX xÅDC1STX ðu2e
Åo°BS fä_llÓBVT @Œ, ØY E21í + 'l203%8ñÙ>SI ð=EW (ñ</string>
<string name="meters_plain">[{&quot;city:&quot;:&quot;Witten&quot;,&quot;clientFirstname:&quot;:&quot;
,&quot;&quot;,&quot;clientLastname:&quot;:&quot;,&quot;contractNumber:&quot;:&quot;12345&quot;
,&quot;zip:&quot;:&quot;58456&quot;,&quot;meterNumber:&quot;:&quot;12345&quot;,&quot;meterReadings
:&quot;:[{&quot;creationTime:&quot;:&quot;Oct 21, 2015 1:31:41 PM&quot;,&quot;date:&quot;:&quot;
Aug 21, 2016 12:00:00 AM&quot;,&quot;name:&quot;:&quot;Sommer&quot;,&quot;value:&quot;:2123000},{&
quot;creationTime:&quot;:&quot;Oct 21, 2015 1:31:41 PM&quot;,&quot;date:&quot;:&quot;Dec 23, 2015
12:00:00 AM&quot;,&quot;name:&quot;:&quot;Test&quot;,&quot;value:&quot;:292929},{&quot;
creationTime:&quot;:&quot;Oct 21, 2015 1:31:41 PM&quot;,&quot;date:&quot;:&quot;Oct 21, 2015
12:00:00 AM&quot;,&quot;name:&quot;:&quot;Winter&quot;,&quot;value:&quot;:1000}]</string>

```

Abbildung 20: Auszug SharedPreferences swd-App

Die ersetzte load-Methode (s. Listing 43) lädt die verschlüsselt gespeicherten Daten.

```
public static List<EnergyMeter> load(final Context context) {
    try {
        final SecureAndroid secureAndroid = new SecureAndroid(context);
        final Gson gson = new Gson();
        final byte[] temp = secureAndroid.retrieve(
            SecureAndroid.SHARED_PREFERENCES, "meters");
        final String jsonString = new String(temp, "UTF-8");
        final TypeToken<List<EnergyMeter>> typeToken = new
            TypeToken<List<EnergyMeter>>(){};
        final Type type = typeToken.getType();
        return gson.fromJson(jsonString, type);
    } catch (CryptoIOHelper.NoKeyMaterialException e) {
        return Lists.newArrayList();
    } catch (Exception e) {
        Log.e(LOG_TAG, "load()", e);
    }
    return Lists.newArrayList();
}
```

Listing 43: load-Methode

Die verschlüsselte JSON-Repräsentation der EnergyMeter-Liste wird geladen, entschlüsselt und in den JSON-String gespeichert. Um aus diesem String wieder die EnergyMeter-Liste zu generieren, muss ein TypeToken der Liste erstellt werden. Ein TypeToken ist eine Hilfsklasse aus der GSON-Bibliothek, die einen Datentyp bestimmt. Die Liste der EnergyMeter kann dann mit Hilfe des Tokens durch Aufruf der Methode `fromJson` aus dem JSON-String generiert werden. Die Methode benötigt den Token um zu wissen, in welches Java-Objekt sie den String überführen soll. Abb. 21 zeigt mit load geladene Zählerstände.



Abbildung 21: Geladene Zählerstände swd-App

◆ **Speicherung Nutzeradresse**

Weil die Verwaltung der Nutzeradresse und der Wohnungsparameter von App zu App unterschiedlich ist, wird die Speicherung dieser Daten nicht in `core`, sondern im App-eigenen Paket `swd_app` gehandhabt. Die `saveUserWasteAdress`-Methode zum Speichern und die `read`-Methode zum Laden der Adresse befinden sich in der Klasse `GsonTool` (s. Anhang B11). Die Methoden arbeiten nach dem gleichen Prinzip wie die Methoden zum Speichern und Laden der Zähler. Aus dem `UserSettings`-Objekt wird die JSON-Repräsentation generiert. Der daraus entstandene String wird in ein Byte-Array überführt und der `encryptAndStore`-Methode übergeben. In der Lade-Methode wird aus den entschlüsselten Bytes wieder der JSON-String und mit der Methode `fromJson` das `UserSettings`-Objekt erzeugt. Die Methoden `saveUserWasteAdress` und `read` werden beispielsweise in `Settings`- und `StreetListFragment` von `swd_app` aufgerufen. Hier werden die Address- und Wohnungsdaten geladen bzw. gespeichert. Eine vollständige Übersicht über Methodenaufrufe mit Beteiligung von `SecureAndroid` findet sich in Tabelle 4 (s. Kap. 4.7.2).

4.7. Produktive Nutzung in eap-Apps

4.7.1. Übergang zu SecureAndroid

Für die Produktiv-Nutzung von `SecureAndroid` in den eap-Apps muss ein Mechanismus für den Übergang zur Nutzung des Frameworks eingebaut werden. Wenn eine App mit `SecureAndroid` erweitert wird, sind zu diesem Zeitpunkt noch unverschlüsselte Daten vorhanden, die übernommen und dann verschlüsselt werden müssen. Danach müssen die unverschlüsselten Datensätze gelöscht werden.

Um dies zu erreichen, wird die ersetzte `load`-Methode (s. Kap. 4.6.3) um eine Ausnahmebehandlung erweitert (s. Listing 44).

```
catch (CryptoIOHelper.NoKeyMaterialException|CryptoIOHelper
        .DataNotFoundException e) {
    final File meterFile = CoreApplication.get(context).getMeterFile();
    final List<EnergyMeter> temp = loadOld(meterFile);
    save(temp, context);
    meterFile.delete();
    return load(context);
}
```

Listing 44: Ausnahmebehandlung für Übergang zu `SecureAndroid`

In diesem `catch`-Block wird der Fall, dass keine verschlüsselt gespeicherten Daten gefunden wurden, behandelt. Dieser Fall tritt ein, wenn ein User zum ersten Mal eine App

mit integrierter SecureAndroid-Bibliothek ausführt, z. B. beim Update von einer alten Version. Falls keine verschlüsselten Daten gefunden werden, wird die alte `load`-Methode (`loadOld`) aufgerufen. Diese Methode lädt die unverschlüsselt gespeicherten Daten oder liefert eine leere Liste, falls keine Daten vorhanden sind. Falls Daten vorhanden sind, werden diese daraufhin sofort verschlüsselt gespeichert und die unverschlüsselten Daten werden gelöscht. Anschließend ruft die `load`-Methode sich selbst auf und gibt die dann verschlüsselt gespeicherten Daten zurück. Abb. 22 zeigt den Datenfluss dieses Vorgangs auf.

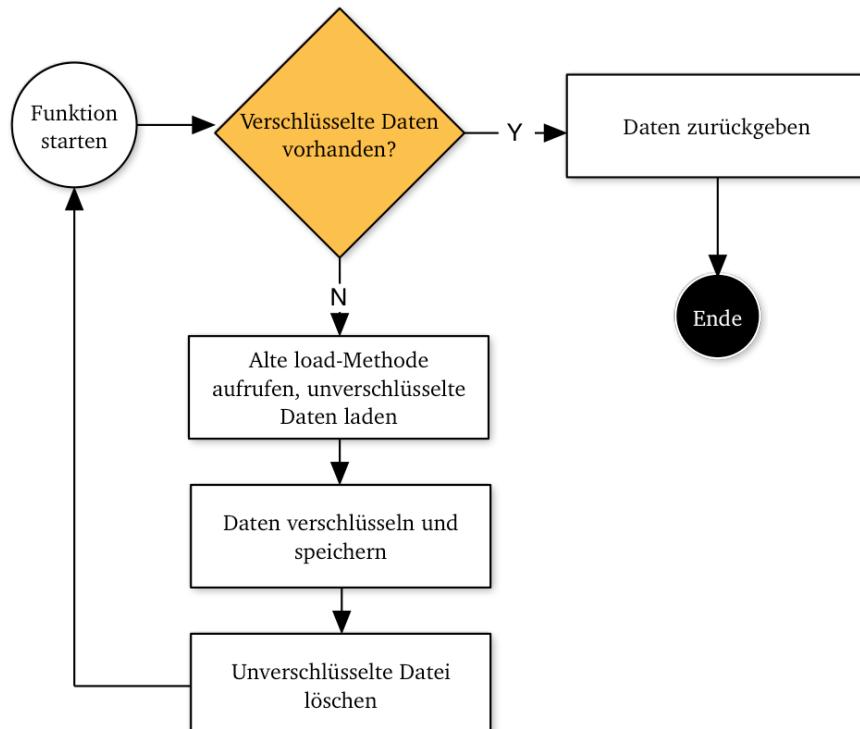


Abbildung 22: Datenfluss App-Update auf Version mit SecureAndroid

Die Methode `read` aus der Klasse `GsonTool` (s. Kap. 4.6.3) wurde analog zu der oben beschriebenen `load`-Methode angepasst (s. Anhang B12). Um die Ladezeit beim erstmaligen Laden der Zähler nach einem App-Update auf eine Version mit SecureAndroid so kurz wie möglich zu halten, wird der Methode `checkAppFirstRunAndShowUpdateInfo` aus der `CoreNavigationDrawerActivity` die in Listing 45 ersichtliche Überprüfung hinzugefügt.

```

if (!(preferences.getBoolean(SECURE_ANDROID_TRANSITION, false))) {
    MeterReader.load(CoreNavigationDrawerActivity.this);
    final SharedPreferences.Editor editor = preferences.edit();
    editor.putBoolean(SECURE_ANDROID_TRANSITION, true);
    editor.apply();
}
  
```

Listing 45: Überprüfung ob Daten bereits verschlüsselt wurden

In diesem Abschnitt wird überprüft, ob die `load`-Methode aus `MeterReader` bereits ausgeführt wurde. Wenn dies der Fall ist, ist der Übergang von unverschlüsselten zu verschlüsselten Zählern und Zählerständen bereits vollzogen, wenn nicht, wird der Übergang vollzogen. Weil die Methode `checkAppFirstRunAndShowUpdateInfo` beim Start einer App aufgerufen wird, hat die Übergangs-Verzögerung keine negative Auswirkung auf das Nutzungserlebnis.

4.7.2. Asynchrones Ver- und Entschlüsseln

Um einem App-Nutzer das bestmögliche Erlebnis zu bieten, sollten rechenintensive Operationen in Hintergrund-Threads ausgeführt werden damit diese den UI-Thread der App nicht blockieren. In den Apps von eap wird dazu das Framework `RxJava`⁷ verwendet. Die Funktionsweise von RxJava ist an das Observer-Pattern angelehnt. Um die Ver- und Entschlüsselungsvorgänge im Hintergrund ablaufen zu lassen, werden in der Klasse `MeterReader` aus dem `core`-package und in den App-spezifischen Klassen `GsonTool` entsprechende RxJava-Funktionen hinzugefügt. Beispielhaft für die Nutzung der Observables aus RxJava ist die Anpassung der Methode `saveUserWasteAddress` aus `GsonTool` (`swd_app`) (s. Anhang B11). Für diese Methode wird eine Anpassung benötigt, so dass über einen booleschen Wert angezeigt wird, ob Daten erfolgreich gespeichert wurden. Die angepasste Methode `saveUserWasteAddressBoolean` findet sich in Listing 46.

```
private static boolean saveUserWasteAddressBoolean(@NonNull final
    UserSettings address, @NonNull final SWDApplication applicationObject)
{
    try {
        final String json = gson.toJson(address);
        SecureAndroid sc = new SecureAndroid(applicationObject
            .getApplicationContext(), 1000);
        sc.encryptAndStore(SecureAndroid.NO_INTEGRITY,
            SecureAndroid.SHARED_PREFERENCES, json.getBytes("UTF-8"),
            SWDApplication.getUserWasteCalendarFileName());
        return true;
    } catch (Exception e) {
        return false;
    }
}
```

Listing 46: `saveUserWasteAddressBoolean`-Methode

Mit Hilfe dieser Methode kann nun die Methode `saveUserWasteAddressAsync` (s. Listing 47) definiert werden.

⁷ <https://github.com/ReactiveX/RxJava>

```

public static Observable<Boolean> saveUserWasteAddressAsync(@NonNull final
    UserSettings address, @NonNull final SWDApplication applicationObject)
{
    return Observable.defer(() -> Observable.just(
        saveUserWasteAddressBoolean(address, applicationObject)));
}

```

Listing 47: saveUserWasteAddressAsync-Methode

saveUserWasteAddressAsync gibt ein Observable vom Typ Boolean zurück. Das Rückgabeobjekt wird erstellt, indem die ursprüngliche saveUserWasteAddressBoolean-Methode mittels Observable.just zu einem Observable-Objekt transformiert wird. Observable.defer stellt sicher, dass das Observable erst erstellt wird, wenn ein Subscriber das Ergebnis von saveUserWasteAddressAsync anfragt. Listing 48 zeigt den Aufruf von saveUserWasteAddressAsync in SettingsFragment.

```

subscription.add(GsonTool.saveUserWasteAddressAsync(userSettings,
    SWDApplication.get(getActivity()))
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Subscriber<Boolean>() {
        @Override
        public void onCompleted() {}

        @Override
        public void onError(Throwable e) {
            if (getActivity() != null) {
                Toast.makeText(getActivity(),
                    R.string.failed_to_save_file,
                    Toast.LENGTH_LONG).show();
            }
        }

        @Override
        public void onNext(Boolean success) {
            if (getActivity() != null) {
                if (success) {
                    Toast.makeText(getActivity(), R.string.file_saved,
                        Toast.LENGTH_LONG).show();
                } else {
                    Toast.makeText(getActivity(),
                        R.string.failed_to_save_file,
                        Toast.LENGTH_LONG).show();
                }
            }
        }
    }));

```

Listing 48: Aufruf saveUserWasteAddressAsync

`saveUserWasteAddressAsync` wird das `UserSettings`-Objekt übergeben. Durch den Aufruf von `subscribeOn(Schedulers.io())` wird der Thread festgelegt, in dem der auszuführende Code ablaufen soll. Der Hintergrund-Thread `io` blockiert nicht den Hauptthread. `observeOn` legt den Thread fest, der auf das Ergebnis wartet. In diesem Fall ist dies der Hauptthread. In `subscribe` wird festgelegt, was bei erfolgreichem Ablauf der Methode geschehen soll. Bei erfolgreicher Speicherung der Daten, wird eine entsprechende Meldung an den Nutzer ausgegeben. Im Fehlerfall wird eine Fehlermeldung ausgegeben.

Auch in der `MeterReader`-Klasse im `core-package` werden die Methoden zum Laden und Speichern von Zählerständen um asynchrone Funktionalitäten erweitert. In `MeterReader` befinden sich somit vier Methoden zum Laden und Speichern von Zählerständen: `load`, `save`, `asyncLoad` und `asyncSave`. In `GsonTool` befinden sich ebensfalls vier Methoden zum Speichern und Laden von Nutzereinstellungen: `read`, `saveUserWasteAddressBoolean`, `asyncRead` und `saveUserWasteAddressAsync`. Insgesamt werden die durch die Implementierung von `SecureAndroid` veränderten Methoden an bis zu 17 Stellen im eap-Code aufgerufen. Dies ist vom jeweiligen App-Umfang abhängig. Tabelle 4 zeigt eine Übersicht dieser Aufrufe. Die Anpassungen im `core-package` (Zähler und Zählerstände) müssen nur einmal vollzogen werden. Für die Aufrufe aus `GsonTool` (Adressdaten) muss jede App separat angepasst werden.

Nutzung von SecureAndroid in eap-Apps

Zähler- und Zählerstände	load	save	asyncLoad	asyncSave
CoreCreateMeterFragment	x	x		
CoreNaviagtionDrawerAct.	x			
MeterReader	x	x		
EnergyMeterAdapter			x	
CoreFragmentMeterOverv.				x
Adressdaten	read	saveUserWasteAddressB.	asyncRead	saveUserWasteAddressAs.
ComparisonFragment	x			
GsonTool	x	x		
StreetListFragment	x		x	x
SettingsFragment			x	x
WasteCalendarFragment			x	
MeterOverviewFragment			x	

Tabelle 4: SecureAndroid-Aufrufe in eap-Apps

4.7.3. Größenänderung der Apps

Eine Anforderung an SecureAndroid ist, dass die Bibliothek die Größe einer App nicht aufblähen sollte (s. Kap. 1.2). Um das Erreichen dieses Ziels zu testen, werden vier eap-Apps auf ihre Größe ohne und mit SecureAndroid überprüft. Tabelle 5 zeigt das Ergebnis des Tests. SecureAndroid hat keine praktische Auswirkung auf die Größe einer App; die Bibliothek ist sehr leichtgewichtig und hat die Zielvorgabe somit erreicht.

Vergleich Paket- und Installationsgröße SecureAndroid			
	Ohne SA	Mit SA	Differenz
SWD APK	5.9 MB	5.9 MB	0 MB
SWD Installation	24.46 MB	24.48 MB	+ 0.02 MB
Schwere APP APK	15.4 MB	15.4 MB	0 MB
Schwere APP Installation	33.20 MB	33.28 MB	+ 0.08 MB
Meine ENNI APK	16.5 MB	16.5 MB	0 MB
Meine ENNI Installation	34.45 MB	34.52 MB	+ 0.07 MB
MainOrt APK	13.6 MB	13.6 MB	0 MB
MainOrt Installation	31.46 MB	31.54 MB	+ 0.08 MB
Energiebündel APK	16.1 MB	16.1 MB	0 MB
Energiebündel Installation	33.83	33.91 MB	+ 0.08 MB

Tabelle 5: Vergleich Paket- und Installationsgrößen mit und ohne SecureAndroid

5. Vergleich mit Alternativen

In diesem Kapitel werden alternative Frameworks und Klassenbibliotheken zur Verschlüsselung unter Android vorgestellt und mit SecureAndroid verglichen. Dabei werden insbesondere die Unterschiede und Verbesserungen von SecureAndroid gegenüber den betrachteten Alternativen herausgestellt.

5.1. Facebook-Conceal

Das von Facebook entwickelte Android-Verschlüsselungsframework *Conceal*⁸ dient in erster Linie zur performanten Verschlüsselung und Speicherung von größeren Datensätzen in Android-Smartphones. Facebook nutzt das Framework zur Verschlüsselung von Bildern auf SD-Karten. Conceal bietet wenige, einfach gestaltete Methoden zur Verschlüsselung und Speicherung an. Allerdings gebraucht Conceal zur Ver- und Entschlüsselung nur einen Schlüssel. Dieser Schlüssel wird *unverschlüsselt* in den SharedPreferences auf dem Smartphone oder Tablet gespeichert. Bei Kompromittierung des Geräts ist der Schlüssel also nicht geschützt.

Conceal wurde ausdrücklich mit dem primären Ziel entwickelt, Daten die auf externen SD-Karten gespeichert werden, vor dem Zugriff von anderen Apps zu schützen. Eigentlich ist dieser Schutzmechanismus in Android bereits implementiert. Anderen Apps kann der Zugriff auf Daten, die auf dem internen Gerätespeicher (SharedPreferences, Datei, SQLite-Datenbank) gespeichert werden, untersagt werden. Dieser Schutz greift allerdings nicht bei externen SD-Karten (s. Praxisphasenbericht Kap. 4.4). Conceal ist somit eine Erweiterung der App-Sandbox auf den externen Speicher. Das Framework ist allerdings nicht darauf ausgelegt, Daten auf einem von einem Angreifer oder von Malware kompromittierten Gerät zu schützen. Wegen des unverschlüsselt abgespeicherten AES-Master-Keys ist dies nicht möglich. Somit ist die Vertraulichkeit von Daten im Gegensatz zu SecureAndroid bei einem Angriff auf das Gerät in *keinster Weise* gegeben. Folglich kann Conceal nicht als wirkliche Alternative für SecureAndroid in Betracht gezogen werden. Zusätzlich ist die Einbindung von Conceal in ein Android-Projekt nicht trivial wie [20] entnommen werden kann.

5.2. SQL-Cipher

SQL-Cipher [35] ist ein Framework zur Verschlüsselung einer SQLite-Datenbank. SQL-Cipher ist kein Android-spezifisches Projekt, kann aber in der Android-Entwicklung benutzt werden. Das Framework bietet in vielen Punkten ähnliche Features wie SecureAndroid:

⁸ <https://github.com/facebook/conceal>

- AES-Verschlüsselung im CBC-Modus
- Der Gebrauch von zufälligen IVs
- Schlüsselgenerierung mittels PBKDF
- Integritätsprüfung per HMAC

Im Gegensatz zu SecureAndroid ist die der PBKDF zugrundliegende Hash-Funktion jedoch immer SHA-1. Eine Möglichkeit für die Nutzung von SHA-256 bietet SQL-Cipher nicht. Des Weiteren arbeitet SQL-Cipher nicht mit Intermediate-Keys und verstößt somit gegen die Grundanforderung an SecureAndroid. Zusätzlich gestaltet sich das Einbinden in ein Android-Projekt als hürdenreich, wie [36] entnommen werden kann. Zuletzt verschlüsselt SQLCipher nur eine SQLite-Datenbank, bietet also keine Möglichkeit, Daten nur zu verschlüsseln und mit diesen Daten dann nach Belieben zu verfahren bzw. Daten direkt in den SharedPreferences oder in eine Datei zu speichern.

5.3. Klassenbibliotheken

Verschiedene Entwickler bieten im Internet kleine Klassenbibliotheken zur Vereinfachung von Verschlüsselung in Android an. Das Mini-Framework *SecurePreferences* [33] ist ein Verschlüsselungswrapper für die SharedPreferences, speichert den AES-Schlüssel allerdings wie Conceal unverschlüsselt ab, wenn kein Nutzernpasswort eingegeben wird. Die Klassenbibliothek *java-aes-crypto* [39] bietet die komfortable Verschlüsselung von z. B. Strings an, aber keine Schlüsselverwaltung.

5.4. Fazit des Vergleichs

Allen hier mit SecureAndroid verglichenen Bibliotheken ist gemein, dass sie nicht nach dem Prinzip des Intermediate-Keys arbeiten. Damit ist die Grundanforderung an eine kryptographische Bibliothek im Rahmen dieser Arbeit nicht erfüllt. Des Weiteren bieten die soeben beschriebenen Bibliotheken keine automatisierte Schlüsselverwaltung mit verschlüsselter Schlüsselspeicherung. Diese Bibliotheken decken für sich immer nur einen spezifischen Teilbereich der Anforderungen an SecureAndroid ab und sind aus diesem Grund dem in dieser Arbeit entwickelten Framework unterlegen. Einzig SecureAndroid bietet automatisiertes Verschlüsseln von Daten und deren Speicherung, ohne sich über die verwendeten kryptographischen Schlüssel und deren Sicherheit Gedanken machen zu müssen. SecureAndroid bietet einem Entwickler gleichzeitig mehr Möglichkeiten, als die hier betrachteten Alternativen. Beispielsweise können Daten direkt als Datei *und/oder* in den SharedPreferences gespeichert werden, dies ist aber kein Muss. Der Entwickler kann die verschlüsselten Daten auch direkt erhalten und dann mit diesen nach seinem Belieben verfahren. Folglich bietet in diesem Vergleich allein SecureAndroid einem Entwickler an, zwischen Einfachheit und komplexeren Möglichkeiten zu entscheiden.

6. Zusammenfassung und Fazit

Diese Arbeit dient dazu, eine Verschlüsselungsbibliothek – SecureAndroid genannt – für das Betriebssystem Android zu schreiben. SecureAndroid soll sichere Verschlüsselung in Android ermöglichen, ohne dass ein Entwickler zur Nutzung der API ein Kryptographie-Experte sein muss. Dazu gehört, dass die Verwaltung der kryptographischen Schlüssel automatisch im Hintergrund geschieht. Des Weiteren soll SecureAndroid keinen Fehler wiederholen, die häufig in der Anwendung von Kryptographie unter Android gemacht werden. Insgesamt besteht das Ziel dieser Arbeit darin, eine Lücke in der Android-Entwicklung zu schließen, da eine einfach zu bedienende und sichere Verschlüsselungsbibliothek mit automatischer Schlüsselverwaltung nach dem Prinzip der Intermediate-Keys vor dieser Arbeit nicht existierte.

Um diese Ziele zu erreichen, wird in SecureAndroid zur Sicherstellung der Vertraulichkeit der Daten der AES-128-Verschlüsselungsalgorithmus in dem Betriebsmodus CBC verwendet. Für jeden Verschlüsselungsvorgang wird ein zufälliger IV erstellt und vor der Verschlüsselung des ersten Blocks mit dem Klartext verknüpft. Zusätzlich sorgt die Verwendung des HMAC-SHA-256-Verfahrens für die Integrität und Authentizität der Daten. Zur Ver- und Entschlüsselung mit AES und für HMAC werden zufällig erstellte Intermediate-Keys verwendet. Diese Intermediate-Keys werden mit einem AES-Master-Key verschlüsselt und abgespeichert. Zusätzlich wird die Integrität und Authentizität der verschlüsselt gespeicherten Intermediate-Keys mit einem MAC-Master-Key sichergestellt. Die Master-Keys werden nicht auf dem Gerät gespeichert, sondern bei jeder Ver- und Entschlüsselung aus einem Passwort generiert. Je nach Verfügbarkeit auf einem Gerät, geschieht dies basierend auf dem SHA-1- bzw. SHA-256-Verfahren mittels einer PBKDF. Das zur Generierung der Master-Keys benötigte Passwort besteht entweder aus einem Hash-Wert von gerätespezifischen Informationen – ist also auf jedem Android-Gerät unterschiedlich – oder wird von dem Entwickler zur Verfügung gestellt und auf dem Gerät als Hash-Wert gespeichert.

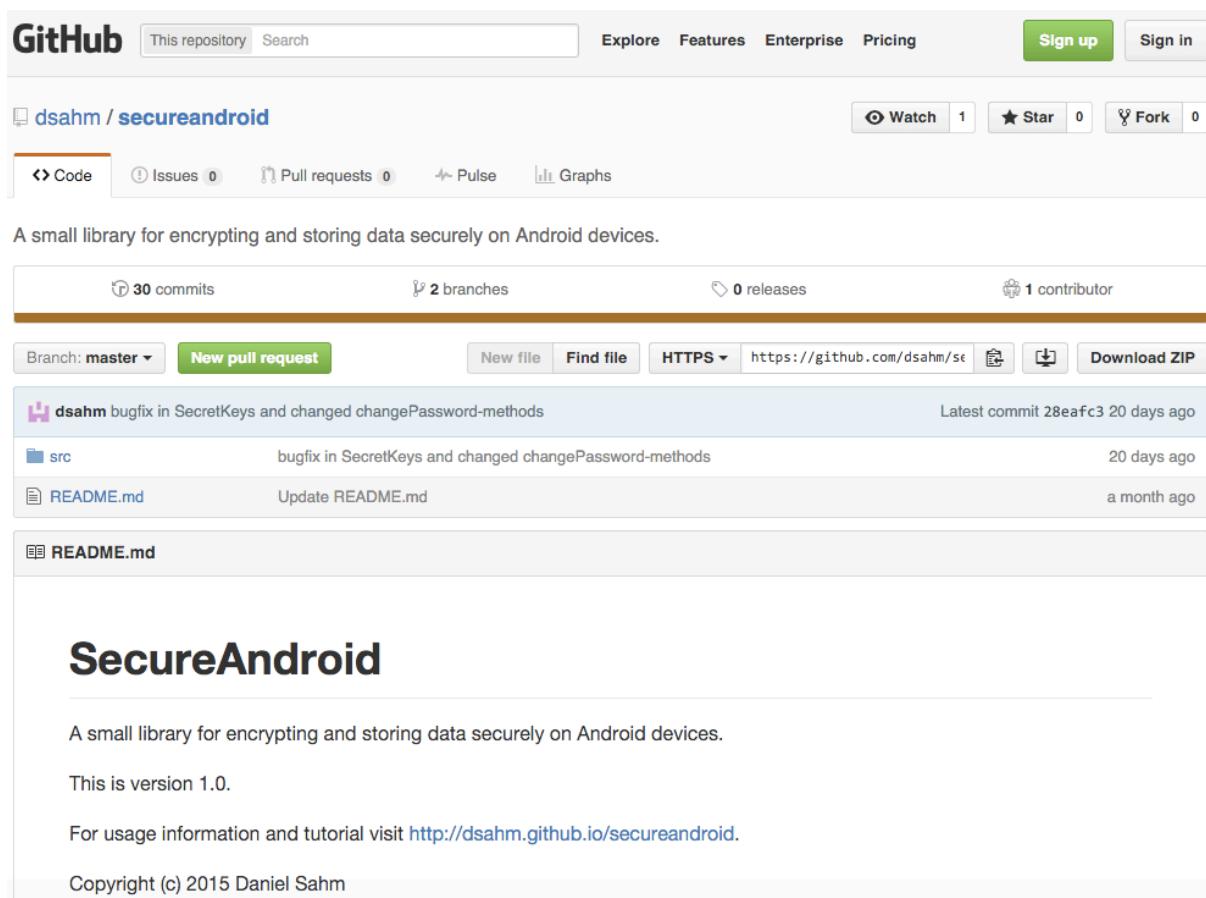
SecureAndroid wird auf drei Arten getestet: Mit Unit-Tests, einer Beispiel-App und in den eap-Apps. Die Unit-Tests testen die Methoden der Bibliothek auf logischer Ebene und überprüfen, ob Exceptions korrekt ausgelöst werden. Die Beispiel-App bietet eine Oberfläche an, die alle Funktionen von SecureAndroid abbildet und nach einer Verschlüsselung den Chiffretext anzeigt. Danach können die Daten wieder entschlüsselt werden. Zusätzlich kann im Dateisystem von Android überprüft werden, ob Daten tatsächlich verschlüsselt wurden. Das Testen mit den eap-Apps dient dazu, die Funktionalitäten von SecureAndroid in einem größeren Zusammenhang zu testen. Zusammenfassend lässt sich feststellen, dass alle Tests erfolgreich abgeschlossen und die für SecureAndroid definierten Ziele in jedem Punkt erreicht werden. Nach dem Vergleich mit alternativen Bibliotheken wird deutlich, dass SecureAndroid ein überlegenes Framework ist, welches alle wichtigen Punkte in der Anwendung von Kryptographie unter Android beachtet und gleichzeitig per Standard sicher und leicht zu bedienen ist. Somit schließt SecureAndroid wie gefordert eine Lücke in der Android-Programmierung.

6.1. Aktueller Projektstatus

◆ Veröffentlichung

Zum Zeitpunkt der Fertigstellung dieser Arbeit ist die Version 1.0 von SecureAndroid auf GitHub veröffentlicht. Zusätzlich zu der GitHub-Seite besteht eine Homepage mit Einführung in das Framework anhand von Code-Beispielen. SecureAndroid ist unter der MIT-Lizenz⁹ veröffentlicht und somit in jeglicher Form von Dritten nutzbar.

Die GitHub-Seite kann unter <https://github.com/dsahm/secureandroid> aufgerufen werden (s. Abb. 23). Die Homepage mit dem Tutorial (s. Abb. 24) und die Javadocs sind von der GitHub-Seite zu erreichen. Die Einbindung in ein Android-Projekt ist trivial wie <http://dsahm.github.io/secureandroid> zu entnehmen ist. Die SecureAndroid-Klassen müssen lediglich in einen Ordner innerhalb der Projektstruktur kopiert werden. Es muss kein externes Repository oder eine andere Abhängigkeit hinzugefügt werden. Die Homepage mit der Dokumentation erklärt die Nutzung von SecureAndroid in einem Android-Projekt anhand von Code-Beispielen. Folglich ist auch die Forderung einer guten Dokumentation von SecureAndroid erfüllt.



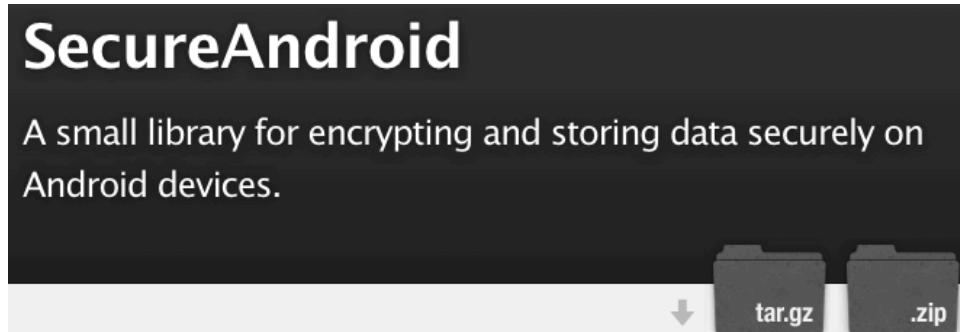
The screenshot shows the GitHub homepage for the repository `dsahm / secureandroid`. At the top, there's a navigation bar with links for Explore, Features, Enterprise, Pricing, Sign up, and Sign in. Below the header, the repository name is displayed with a star icon and a fork icon. A summary bar shows 30 commits, 2 branches, 0 releases, and 1 contributor. Below this, a list of recent commits is shown:

- dsahm** bugfix in SecretKeys and changed changePassword-methods - Latest commit 28eafc3 20 days ago
- src** bugfix in SecretKeys and changed changePassword-methods - 20 days ago
- README.md** Update README.md - a month ago

At the bottom of the page, there's a section titled "SecureAndroid" with a brief description: "A small library for encrypting and storing data securely on Android devices." It also mentions that it's version 1.0 and provides a link to the documentation at <http://dsahm.github.io/secureandroid>.

Abbildung 23: GitHub-Homepage SecureAndroid

⁹ <https://opensource.org/licenses/MIT>



Welcome to SecureAndroid

SecureAndroid is a small library that will help you to easily encrypt, decrypt and store data securely on Android devices. The library was designed with ease-of-use in mind. You do not have to worry about implementation details of cryptographic algorithms but simply use the provided API. SecureAndroid uses AES encryption in CBC mode with padding according to PKCS5. For data integrity HMacSHA256 or HMacSHA1 (according to the availability on the device) is used. For key derivation from passwords the PBKDF2WithHmacSHA256/1 (according to the availability on the device) is employed. SecureAndroid abstracts cryptographic implementation details and i/o-operations away so that you can concentrate on your app logic.

How to add SecureAndroid to a project

To use SecureAndroid, simply download the zip or tar.gz file and extract it. In your Android Studio project move to the java folder of your module:

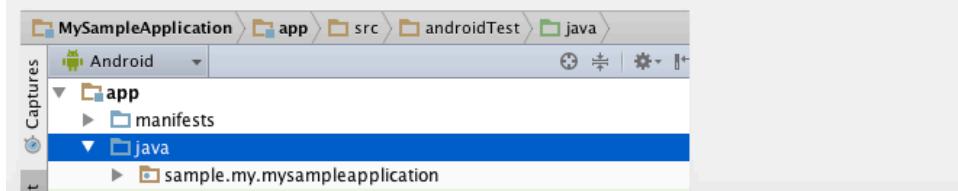


Abbildung 24: Dokumentation SecureAndroid

◆ Nutzung im Unternehmen

Mit Stand 16.02.2016 wird SecureAndroid in allen eap-Apps genutzt:

- Energiebündel (Stadtwerke Remscheid)
- ENTEGA Regional (Stadtwerke Darmstadt und Mainz)
- Käppsele (Stadtwerke Karlsruhe)
- Leipziger (Stadtwerke Leipzig)
- MainOrt (Stadtwerke Frankfurt am Main)
- MeinLÜBECK (Stadtwerke Lübeck)
- Mein Quadrat (Stadtwerke Mannheim)
- N-Ergie (Stadtwerke Nürnberg)
- Niederrhein APPtuell (Stadtwerke Mörs)
- Schwerte App (Stadtwerke Schwerte)
- SWD (Stadtwerke Düsseldorf)
- WI+WAS (Stadtwerke Wiesbaden)

All diese Apps sind im Google-Playstore verfügbar. SecureAndroid übernimmt das Verschlüsseln und Speichern von Zählern und den zugehörigen Zählerständen sowie von Nutzerdaten wie Adresse und Wohnungsparametern. Nach erfolgreichen Tests läuft die Bibliothek zuverlässig und stabil.

6.2. Ausblick

SecureAndroid ist darauf ausgelegt, von der Android-Entwickler-Community genutzt zu werden. Noch ist nicht absehbar, wie gut das Framework aufgenommen wird und in größeren Entwicklungszusammenhängen funktioniert. Aus diesem Grund gibt es auf GitHub die Möglichkeit, Bugs zu melden. Diese Rubrik findet sich unter dem Menüpunkt *Issues* auf der GitHub-Seite des SecureAndroid-Projekts. Aufgrund der hohen Komplexität heutiger Software-Projekte ist es nicht unwahrscheinlich, dass Szenarien auftreten werden, in denen die Logik von SecureAndroid nicht (gut) funktioniert. Auch in diesem Fall können Entwickler über GitHub Kontakt mit dem Autor dieser Arbeit aufnehmen, um Verbesserungsvorschläge zu unterbreiten. Im Zuge der Evolution des Android-Betriebssystems wird es auch dazu kommen, dass Anpassungen an SecureAndroid vorgenommen werden müssen. Wenn beispielsweise zukünftig besser geeignete AES-Betriebsmodi standardmäßig auf jedem Android-Gerät angeboten werden, könnte ein Wechsel zu einem dieser Modi sinnvoll sein. Für den Fall, dass Google dafür sorgen sollte, dass sich die Verfügbarkeit von kryptographisch sicheren Hash-Funktionen in Kombination mit der PBKDF verbessert, ist SecureAndroid bereits gerüstet (s. Kap. 3.2).

Literaturverzeichnis

- [1] „Activities | Android Developers“. Zugegriffen 29. Oktober 2015. <http://developer.android.com/guide/components/activities.html>.
- [2] „A Lesson In Timing Attacks (or, Don't use MessageDigest.equals) | codahale.com“. Zugegriffen 4. Januar 2016. <http://codahale.com/a-lesson-in-timing-attacks/>.
- [3] „Android and iOS Squeeze the Competition, Swelling to 96.3% of the Smartphone Operating System Market for Both 4Q14 and CY14, According to IDC“. [www.idc.com](http://www.idc.com/getdoc.jsp?containerId=prUS25450615). Zugegriffen 28. Oktober 2015. <http://www.idc.com/getdoc.jsp?containerId=prUS25450615>.
- [4] „Android Context“. Zugegriffen 14. Oktober 2015. <http://developer.android.com/reference/android/content/Context.html>.
- [5] Android Developers. „Some SecureRandom Thoughts | Android Developers Blog“. Zugegriffen 13. Oktober 2015. <http://android-developers.blogspot.de/2013/08/some-securerandom-thoughts.html>.
- [6] Baraniuk, Chris. „The number glitch that can lead to catastrophe“. Zugegriffen 5. November 2015. <http://www.bbc.com/future/story/20150505-the-numbers-that-lead-to-disaster>.
- [7] „BDSG“. Zugegriffen 28. Oktober 2015. http://www.gesetze-im-internet.de/bdsg_1990/.
- [8] Becker und Pant. 2014. *Android 4.4*. 3. Aufl. Wieblinger Weg 17, 69123 Heidelberg: dpunkt.verlag GmbH.
- [9] Bellare und Namprempre. Juli 2007. „Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm“.
- [10] Albrecht, Beutelspacher, Neumann und Schwarzpaul. 2005. *Kryptografie in Theorie und Praxis*. 1. Aufl. Wiesbaden: Vieweg & Sohn Verlag/GWV Fachverlage GmbH.
- [11] „BSI für Bürger: Passwörter“. Zugegriffen 23. Oktober 2015. https://www.bsi-fuer-buerger.de/BSIFB/DE/MeinPC/Passwoerter/passwoerter_node.html.

- [12] Buhr, Sarah. „Forrester: Tablet Sales Have Plateaued But There's A Future In Business“. *TechCrunch*. Zugegriffen 28. Oktober 2015. <http://social.tech-crunch.com/2015/07/12/forrester-tablet-sales-have-plateaued-but-theres-a-future-in-business/>.
- [13] Bundesamt für Sicherheit in der Informationstechnik. 2015. *Kryptographische Verfahren: Empfehlungen und Schlüssellängen*.
- [14] Chenda Ngak CBS News January. „The 25 most common passwords of 2013“. Zugegriffen 13. Oktober 2015. <http://www.cbsnews.com/news/the-25-most-common-passwords-of-2013/>.
- [15] „China's Unofficial Android App Stores are Malware Minefields“. *Tech in Asia*. Zugegriffen 29. Oktober 2015. <https://www.techinasia.com/china-android-app-stores-malware/>.
- [16] Eckert, Claudia. 2012. *IT-Sicherheit*. 7. Aufl. Rosenheimer Str. 145, 81671 München: Oldenbourg Wissenschaftsverlag GmbH.
- [17] Back und Edquist. 2014. „Strong(er) Randomness“. *Project Report for Information Security Course*.
- [18] Brumley, Egele, Fratantonio, Kruegel und Manuel. 2013. „An Empirical Study of Cryptographic Misuse in Android Applications“. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 73–84. CCS '13. New York, NY, USA: ACM. doi:10.1145/2508859.2516693.
- [19] „Encrypting strings in Android: Let's make better mistakes < Tozny“. Zugegriffen 28. Oktober 2015. <http://tozny.com/blog/encrypting-strings-in-android-lets-make-better-mistakes/>.
- [20] „facebook/conceal“. *GitHub*. Zugegriffen 11. Februar 2016. <https://github.com/facebook/conceal>.
- [21] „IDC: Smartphone OS Market Share“. *www.idc.com*. Zugegriffen 6. Oktober 2015. <http://www.idc.com/prodserv/smartphone-os-market-share.jsp>.
- [22] Jeff. „Guess why we're moving to 256-bit AES keys“. *AgileBits Blog*. Zugegriffen 28. Oktober 2015. <https://blog.agilebits.com/2013/03/09/guess-why-were-moving-to-256-bit-aes-keys/>.

- [23] McConnell, Steven C. 2004. *Code Complete*. Bd. 24. One Microsoft Way, Redmond, Washington 98052-6399: Microsoft Press.
- [24] Mohd. Ehmer Kahn und Frameena Kahn. 2014. „Importance of Software Testing in Software Development Life Cycle". *International Journal of Computer Science Issues* Vol. 11 (2): 120.
- [25] „Orientierungshilfe zu den Datenschutzanforderungen an App-Entwickler und App-Anbieter". 2014. Bayerisches Landesamt für Datenschutzaufsicht.
- [26] „Over 50% of Android Devices Have Unpatched Security Holes". *The Next Web*. Zugegriffen 29. Oktober 2015. <http://thenextweb.com/google/2012/09/13/new-research-shows-50-android-devices-worldwide-unpatched-vulnerabilities/>.
- [27] „OWASP Password Storage Cheat Sheet". Zugegriffen 13. Oktober 2015. https://www.owasp.org/index.php>Password_Storage_Cheat_Sheet.
- [28] „PKCS #5: Password-Based Cryptography Specification Version 2.0". Zugegriffen 13. Oktober 2015. <https://www.ietf.org/rfc/rfc2898.txt>.
- [29] Isberner, Mettler, Raman, Vishwanath und Zhang. 2015. „Cryptographic Vulnerabilities in Android Applications". *FireEye*. Zugegriffen 9. Februar 2016. https://www.fireeye.com/blog/threat-research/2015/01/cryptographic_vulner.html.
- [30] Rodewig und Wagner. 2015. *Apps programmieren für iPhone und iPad*. 3. Aufl. Bonn: Galileo Press.
- [31] Ryan Whitwam. „How Bitcoin thieves used an Android flaw to steal money, and how it affects everyone else". *ExtremeTech*. Zugegriffen 13. Oktober 2015. <http://www.extremetech.com/computing/164134-how-bitcoin-thieves-used-an-android-flaw-to-steal-money-and-how-it-affects-everyone-else>.
- [32] „Schneier on Security - A Really Good Article on How Easy it Is to Crack Passwords“. Zugegriffen 9. Januar 2016. https://www.schneier.com/blog/archives/2013/06/a_really_good_a.html.
- [33] „scottyab/secure-preferences". *GitHub*. Zugegriffen 29. Oktober 2015. <https://github.com/scottyab/secure-preferences>.
- [34] Security, heise. „Cracker-Bremse". *Security*. Zugegriffen 15. Oktober 2015. <http://www.heise.de/security/artikel/Passwoerter-unknackbar-speichern-1253931.html?artikelseite=2>.

[35] „SQL-Cipher. <https://www.zetetic.net/sqlcipher/>“. Zugegriffen 30. September 2015.

[36] „SQL-Cipher for Android. <https://www.zetetic.net/sqlcipher/sqlcipher-for-android/>“. Zugegriffen 30. September 2015.

[37] Svajcer, Vanja. 2014. „Sophos Mobile Security Threat Report“.

[38] „The ‚Stagefright‘ exploit: What you need to know“. *Android Central*. Zugegriffen 29. Oktober 2015. <http://www.androidcentral.com/stagefright>.

[39] „tozny/java-aes-crypto“. *GitHub*. Zugegriffen 29. Oktober 2015. <https://github.com/tozny/java-aes-crypto>.

[40] „Worldwide Tablet Shipments Experience First Year-Over-Year Decline in the Fourth Quarter While Full Year Shipments Show Modest Growth, According to IDC“. [www.idc.com](http://www.idc.com/getdoc.jsp?containerId=prUS25409815). Zugegriffen 28. Oktober 2015. <http://www.idc.com/getdoc.jsp?containerId=prUS25409815>.

Anhang

Teil A: Diagramme

A1: Threat-Model

Betrachtet werden Angriffe welche die Vertraulichkeit, Authentizität und Integrität von auf dem Gerät abzuspeichernden/gespeicherten Daten direkt und indirekt bedrohen.						
Element A	Element B	ID	Bedrohung	Ablöse	Damag	Potential
					Reproducibility	Affected Users
App ohne kryptographische Schutzmaßnahmen.						
User	Activity	1.1	User rootet Gerät und öffnet es für Schadcode.	Bei Programmstart abfragen, ob Gerät gerootet wurde und wenn ja, die relevanten Daten löschen.	3	2
User	Activity	1.1	User installiert bewusst oder unbewusst Software die Schadcode enthält oder fängt sich einen Exploit ein. Schadsoftware versucht, auf dem Gerät gespeicherte Daten auszulesen.	Daten nur verschlüsselt speichern.	3	2
User	Activity	1.1	Ein User verschafft sich unbefugterweise Zugriff auf ein Gerät und will Daten auslesen. Datentransfer wird mitgeschnitten.	Daten verschlüsseln.	3	1
Activity	Datastore	1.2	Angreifer manipuliert Nachricht.	Daten auf Integrität überprüfen.	2	1
Datastore	Activity	1.2			1	2
App wurde um kryptographische Schutzmaßnahmen erweitert.						
User	Activity	1.1	User gibt schwaches Passwort ein, daher schwache kryptographische Schlüssel.	Starke Password-Based-Key-Derivation Funktion mit vielen Iterationen nutzen.	2	3
User	Activity	1.1	User rootet Gerät und öffnet es für Schadcode.	Bei Programmstart abfragen, ob Gerät gerootet wurde und wenn ja, die relevanten kryptographischen Schlüssel löschen um die Daten unlesbar zu machen.	3	2
Activity	Kryptolayer	1.2	Schlüsselmaterial liegt im Arbeitsspeicher und könnte ausgelesen werden.	Schlüssel so kurz wie möglich im Arbeitsspeicher halten.	3	1
Activity	Kryptolayer	1.2	PBKDF2-Funktion hat aus Performance-Gründen nicht genug Iterationen und führt somit zu schwachen kryptographischen Schlüsseln.	Untergrenze für Anzahl Iterationen definieren.	3	2
Kryptolayer	Datastore	1.3	Verschlüsselte Daten werden vor dem Speichern bzw. nach dem Laden manipuliert.	Nach verschlüsseln einen MAC erstellen. Vor dem entschlüsseln mit diesem MAC die Integrität überprüfen.	1	2
Datastore	Kryptolayer	1.3	AES-CBC-Verschlüsselung wegen statischer IVs schwach.	IV für jeden Verschlüsselungsvorgang neu und zufällig generieren.	2	3
Activity	Kryptolayer	1.2	Passwörter können durch Rainbowtables extrahiert werden.	Passwörter vor Hashen mit Salt verknüpfen.	2	3
Activity	Kryptolayer	1.2	Schlüssel kann ausgelesen weil er im Code gespeichert ist oder im Klartext im Dateisystem liegt.	Schlüssel gar nicht oder nur verschlüsselt auf dem Gerät speichern.	3	3
Activity	Kryptolayer	1.2	CSPRNG liefert keine kryptographisch sicheren Zufallszahlen.	Nur CSPRNGs nutzen welche nachweislich kryptographisch sichere Zufallszahlen liefern.	2	1
Activity	Kryptolayer	1.2	Schwacher Betriebsmodus von Verschlüsselungsalgorithmen führt zu angreifbarer Verschlüsselung.	Sichere Betriebsmodi verwenden (z. B. AES-GCM).	2	2
Kryptolayer	Datastore	1.3	Timing-Angriff auf MAC-Verifikation.	Vergleichsmethoden mit fester Laufzeit verwenden.	1	2
Activity	Kryptolayer	1.2	Integrität der Daten durch schwache Hash-Funktion zur Berechnung des MACs in Gefahr.	Nur sichere Hash-Funktionen wie SHA-256 verwenden.	2	1
Activity	Kryptolayer	1.2	Verschlüsselung mit AES-CBC angreifbar weil statische IVs verwendet werden.	IVs immer zufällig erstellen.	2	3

Teil B: Programmcode

B1: createAndStoreAndGetKeyDataWithoutMAC/getKeyDataWithoutMAC

```
/**
 * Private method that creates the AES master-key and stores its salt value. Then
 * the intermediate key
 * is generated and stored in the SharedPreferences. The key then is loaded and
 * returned to the caller.
 * @param password      The password from which the master key will be derived.
 * @return              The AES intermediate-key used for data encryption and
 * decryption.
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 */
private SecretKey createAndStoreAndGetKeyDataWithoutMAC (char[] password) throws
GeneralSecurityException, CryptoIOHelper.DataNotAvailableException,
CryptoIOHelper.IntegrityCheckFailedException {
// Hash the password with salt and get the hashed password and the salt
final PasswordCrypto.HashedPasswordAndSalt hashedPasswordAndSalt =
    passwordCrypto.hashPassword(password);
// Generate master AES key
final AESCrypto.SaltAndKey aesMasterKeyAndSalt = aesCrypto
    .generateRandomAESKeyFromPasswordGetSalt(password);
// Store the salt values used to generate the aes-key
storeRootSaltDataWithoutMAC(aesMasterKeyAndSalt.getSalt());
// Generate the intermediate aes key and encrypt it with the master key
AESCrypto.CipherIV intermediateAESCipherIV = aesCrypto.encryptAES(
    aesCrypto.generateRandomAESKey().getEncoded(), aesMasterKeyAndSalt
    .getSecretKey());
// Store the hashed password+salt, the intermediate key+iv and the mac-key+iv
storePasswordAndIntermediateKeyCipherIVWithoutMAC(hashedPasswordAndSalt,
    intermediateAESCipherIV);
// Load the stored keys to check if save was successful
intermediateAESCipherIV = aesCrypto.getCipherAndIVFromSharedPref(
    IMEDIATE_KEY_DATA, AES_INTERMEDIATEKEY_ALIAS);
// to check if the key and mac material where generated and stored correctly
// Get the raw key material
final byte[] rawAES = aesCrypto.decryptAES(intermediateAESCipherIV.getCipher(),
    intermediateAESCipherIV.getIV(), aesMasterKeyAndSalt.getSecretKey());
// Generate the secret-key objects and return them
return new SecretKeySpec(rawAES, 0, rawAES.length, AES);
}

/**
 * Gets the formerly created and stored key data. Returns the AES intermediate key
 * used for encryption and decryption.
 *
 * @param password      The password from which the master key will be derived.
 * @return              The AES intermediate key used for data encryption and
 * decryption.
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 * @throws CryptoIOHelper.NoKeyMaterialException
 */
private SecretKey getKeyDataWithoutMAC(char[] password) throws
CryptoIOHelper.NoKeyMaterialException, CryptoIOHelper
    .IntegrityCheckFailedException, CryptoIOHelper.DataNotAvailableException,
GeneralSecurityException, CryptoIOHelper.WrongPasswordException {
// Get the formerly hashed password and its salt value from Shared Preferences
try {
    final PasswordCrypto.HashedPasswordAndSalt hashedPasswordAndSalt =
        passwordCrypto.getHashedPasswordAndSaltSharedPref(PASSWORD_ALIAS,
            PASSWORD_HASH_ALIAS_WITHOUT_MAC, PASSWORD_SALT_ALIAS_WITHOUT_MAC);
    // Check whether the hash of the given password+salt equals the hash of the
}
```

```

// stored password
if (passwordCrypto.checkPassword(password, hashedPasswordAndSalt
    .getHashedPassword(), hashedPasswordAndSalt.getSalt())) {
    // Load the salt value for generating the master aes key
    try {
        final byte[] aesSalt = cryptoIOHelper
            .loadFromSharedPrefBase64(KEY_DATA_ALIAS,
                AES_MASTERKEY_WITHOUT_MAC_SALT_ALIAS);
        // Load the encrypted intermediate key
        final AESCrypto.CipherIV intermediateAESCipherIV =
            aesCrypto.getCipherAndIVFromSharedPref(IMEDIATE_KEY_DATA,
                AES_INTERMEDIATEKEY_WITHOUT_MAC_ALIAS);
        // generate Master AES-Key
        final SecretKey aesMasterKey = aesCrypto
            .generateAESKeyFromPasswordSetSalt(password, aesSalt);
        // Extract the raw aes key data
        final byte[] aes = aesCrypto.decryptAES(
            intermediateAESCipherIV.getCipher(),
            intermediateAESCipherIV.getIv(), aesMasterKey);
        // Return the aes intermediate key
        return new SecretKeySpec(aes, 0, aes.length, AES);
    } catch (CryptoIOHelper.DataNotAvailableException e) {
        throw new CryptoIOHelper.NoKeyMaterialException(
            NO_KEYMATERIAL_MSG);
    }
} else {
    // If password check failed, throw WrongPasswordException
    throw new CryptoIOHelper.WrongPasswordException(WRONG_PASSWORD);
}
catch (CryptoIOHelper.DataNotAvailableException e) {
    throw new CryptoIOHelper.NoKeyMaterialException(NO_KEYMATERIAL_MSG);
}
}

```

B2: generateRandomMACKeyFromPasswordGetSalt

```

/***
 * Generates a MAC-Key from the given password.
 *
 * @param password      The password.
 * @return              The SaltAndKey object.
 * @throws GeneralSecurityException
 */
protected SaltAndKey generateRandomMACKeyFromPasswordGetSalt(char[] password)
    throws GeneralSecurityException {
    fixPrng();
    final byte[] salt = super.generateRandomBytes(PBE_SALT_LENGTH_BYTE);
    final KeySpec keySpec = new PBEKeySpec(password, salt, PBE_ITERATIONS,
        MAC_128);
    final SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(
        PBE_ALGORITHM);
    final byte[] temp = keyFactory.generateSecret(keySpec).getEncoded();
    return new SaltAndKey(new SecretKeySpec(temp, MAC_INSTANCE), salt);
}


```

B3: hashPasswordWithSalt

```

/***
 * Hashes a password with the PBKDF2WithHmacSHA512 or the PBKDF2WithHmacSHA1 algo-
 * rithm (depending on the availability
 * on the present platform) with the given salt.
 * @param password      The password.
 * @param salt           The salt.
 * @return              The hashed password.
 * @throws GeneralSecurityException
 */
public byte[] hashPasswordWithSalt (char[] password, byte[] salt) throws


```

```

GeneralSecurityException {
    // Instantiate key specifications with desired parameters
    final KeySpec keySpec = new PBEKeySpec(password, salt, PBE_ITERATIONS,
        KEY_LENGTH);
    // Instantiate key factory with the desired PBE-Algorithm
    final SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(
        PBE_ALGORITHM);
    // Return the hashed password
    return keyFactory.generateSecret(keySpec).getEncoded();
}

```

B4: generateRandomAESKeyfromPasswordGetSalt

```

/***
 * Generates an 128 Bit long AES-Key from the given password.
 *
 * @param password      The password.
 * @return              The AES-Key and the salt.
 * @throws GeneralSecurityException
 */
protected SaltAndKey generateRandomAESKeyFromPasswordGetSalt(char[] password)
    throws GeneralSecurityException {
    fixPrng();
    // Generate random salt
    final byte [] salt = super.generateRandomBytes(PBE_SALT_LENGTH_BYTE);
    // Specifiy Key parameters
    final KeySpec keySpec = new PBEKeySpec(password, salt, PBE_ITERATIONS,
        AES_128);
    // Load the key factory
    final SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(
        PBE_ALGORITHM);
    // Generate random sequence for the key
    final byte[] temp = keyFactory.generateSecret(keySpec).getEncoded();
    // Return new key and salt the key was created with
    return new SaltAndKey(new SecretKeySpec(temp, AES_INSTANCE), salt);
}

```

B5: changePasswordPrivate

```

/***
 * Private method to change the password.
 *
 * @param oldPassword The old password.
 * @param newPassword The new password.
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 */
private boolean changePasswordPrivate(char [] oldPassword, char [] newPassword)
    throws Exception {
    boolean dataNotAvailable = false;
    try {
        final PasswordCrypto.HashedPasswordAndSalt hashedPasswordAndSalt =
            passwordCrypto.getHashedPasswordAndSaltSharedPref(
                PASSWORD_ALIAS, PASSWORD_HASH_ALIAS, PASSWORD_SALT_ALIAS);
        if (passwordCrypto.checkPassword(oldPassword, hashedPasswordAndSalt
            .getHashedPassword(), hashedPasswordAndSalt.getSalt())) {
            wipeKey();
            createAndStoreAndGetKeyData(newPassword);
        } else {
            throw new CryptoIOHelper.WrongPasswordException(
                WRONG_PASSWORD);
        }
    } catch (CryptoIOHelper.DataNotAvailableException e) {
        dataNotAvailable = true;
    }
    try {

```

```

final PasswordCrypto.HashedPasswordAndSalt hashedPasswordAndSalt =
    passwordCrypto.getHashedPasswordAndSaltSharedPref(
        PASSWORD_ALIAS, PASSWORD_HASH_ALIAS_WITHOUT_MAC,
        PASSWORD_SALT_ALIAS_WITHOUT_MAC);
if (passwordCrypto.checkPassword(oldPassword, hashedPasswordAndSalt
    .getHashedPassword(), hashedPasswordAndSalt.getSalt())) {
    wipeKey();
    createAndStoreAndGetKeyDataWithoutMAC(newPassword);
} else {
    throw new CryptoIOHelper.WrongPasswordException(
        WRONG_PASSWORD);
}
} catch (CryptoIOHelper.DataNotAvailableException e) {
    if (dataNotAvailable) {
        throw new CryptoIOHelper.DataNotAvailableException(
            NO_PASSWORD);
    } else {
        return true;
    }
}
return true;
}

```

B6: encryptAndStore

```

/**
 * Encrypts the given plaintext and stores it on the device under the provided
 * alias.
 * @param mode           The storage mode. SecureAndroid.Mode.SHARED_PREFERENCES and
 * SecureAndroid.Mode.FILE are possible.
 * @param plaintext      The plaintext as byte array.
 * @param alias          The alias under which the data will be stored.
 * @throws IOException
 * @throws CryptoIOHelper.WrongPasswordException
 * @throws GeneralSecurityException
 * @throws CryptoIOHelper.DataNotAvailableException
 * @throws CryptoIOHelper.IntegrityCheckFailedException
 * @throws CryptoIOHelper.NoKeyMaterialException
 * @throws GeneralSecurityException
 * @throws IllegalArgumentException
 */
public void encryptAndStore(Integrity integrity, Mode mode, byte[] plaintext,
String alias) throws CryptoIOHelper.NoKeyMaterialException, CryptoIOHelper
    .IntegrityCheckFailedException, IOException, CryptoIOHelper
    .WrongPasswordException, GeneralSecurityException,
CryptoIOHelper.DataNotAvailableException {
switch (integrity) {
    case INTEGRITY:
        // Save the encrypted data under the given alias
        saveUserCipherMACIV(mode, encrypt(integrity, plaintext,
            getAutoPassword().toCharArray()), alias);
        break;
    case NO_INTEGRITY:
        saveUserCipherIV(mode, encrypt(integrity, plaintext,
            getAutoPassword().toCharArray()), alias);
        break;
    default:
        throw new IllegalArgumentException(WRONG_INTEGRITY_MODE_EXCEPTION);
}
}

```

B7: deleteCipherAndIVFromSharedPref/deleteCipherAndIVFile

```

/**
 * Deletes the submitted cipherIV object from the SharedPref submitted under
 * spAlias.
 *
 * @param spAlias         The alias for the SharedPref.
 * @param cipherIVAlias   The alias for the cipherIV object.

```

```

/*
protected void deleteCipherAndIVFromSharedPref(String spAlias, String
cipherIvAlias) {
    super.deleteFromSharedPref(spAlias, cipherIvAlias + CIPHER_PART);
    super.deleteFromSharedPref(spAlias, cipherIvAlias+ IV_PART);
}

/**
 * Deletes the files saved under the given alias for a ciperIv object.
 *
 * @param filename The alias.
 * @throws CryptoIOHelper.DataNotAvailableException
 *
 */
protected void deleteCipherAndIVFile(String filename) throws
DataNotAvailableException {
    super.deleteFile(filename + CIPHER_PART);
    super.deleteFile(filename + IV_PART);
}

```

B8: deleteFromSharedPref/deleteFile/deleteSharedPref

```

/**
 * Deletes the alias from specified SharedPref file.
 *
 * @param spAlias The SharedPreferences alias.
 * @param alias The alias for the file to be deleted.
 */
protected void deleteFromSharedPref(String spAlias, String alias) {
    final SharedPreferences sharedPreferences = context
        .getSharedPreferences(spAlias, Context.MODE_PRIVATE);
    final SharedPreferences.Editor spEditor = sharedPreferences.edit();
    spEditor.remove(alias);
    spEditor.apply();
}

/**
 * Deletes the specified file.
 *
 * @param filename The filename to be deleted.
 * @throws CryptoIOHelper.DataNotAvailableException
 */
protected void deleteFile (String filename) throws DataNotAvailableException {
    final String dir = context.getFilesDir().getAbsolutePath();
    final File file = new File(dir, filename);
    final boolean deleted = file.delete();
    if (!deleted) {
        throw new DataNotAvailableException(DATA_NOT_AVAILABLE);
    }
}

/**
 * Deletes the specified SharedPref file.
 * @param alias The alias for the SharedPref to be deleted.
 */
public void deleteSharedPref(String alias) {
    final SharedPreferences sharedPreferences = context.getSharedPreferences(alias,
        Context.MODE_PRIVATE);
    final SharedPreferences.Editor spEditor = sharedPreferences.edit();
    spEditor.clear();
    spEditor.apply();
}

```

B9: hashPerformanceTest

```

/**
 * Checks the performance of the device when running PBKD.
 *

```

```

* @param iterations The iterations used for first performance check
* @param minIterations Minimum iterations.
* @return The optimal iteration count for good performance.
* @throws NoSuchAlgorithmException
* @throws InvalidKeySpecException
* @throws CryptoI0Helper.NoAlgorithmAvailableException
*/
protected long hashPerformanceTest(int iterations, int minIterations) throws
    NoSuchAlgorithmException, InvalidKeySpecException,
    NoAlgorithmAvailableException {
    checkPerformanceAlgorithmsLength();
    // Performance check start
    final long startTime = System.currentTimeMillis();
    // Instantiate key specifications with desired parameters
    final byte[] salt = generateRandomBytes(PBE_SALT_LENGTH_BYTE);
    final KeySpec keySpec = new PBEKeySpec("testpassword".toCharArray(), salt,
        iterations, KEY_LENGTH);
    final SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(
        PBE_ALGORITHM);
    keyFactory.generateSecret(keySpec).getEncoded();
    // Performance check
    final long stopTime = System.currentTimeMillis();
    final long elapsedTime = stopTime - startTime;
    Log.i("elapsed time", String.valueOf(elapsedTime));
    final long returnIterations = (10000/elapsedTime) * ITERATION_BASE;
    if (minIterations < 1000) {
        minIterations = 1000;
    }
    if (returnIterations < 3*minIterations) {
        return 3*minIterations;
    } else {
        return returnIterations;
    }
}

```

B10: Ersetzte Methoden aus SWD-App

```

public static void save(final List<EnergyMeter> values, final File file) {
    try {
        final ObjectMapper theMapper = new ObjectMapper();
        theMapper.writeValue(file, values);
    } catch (final Exception e) {
        Log.e(LOG_TAG, "save()", e);
    }
}

public static List<EnergyMeter> load(final File file) {
    if (file.exists()) {
        try {
            final InputStream theStream = new FileInputStream(file);
            return loadStream(theStream);
        } catch (final Exception anException) {
            IO.reportException(anException);
            Log.e(LOG_TAG, "load()", anException);
        }
    }
    return Lists.newArrayList();
}

/**
 * Returns either an empty or the filled object when created
 *
 * @param application
 * @return
 * @throws java.io.IOException
 */
public static UserSettings read(@NotNull final SWDApplication application)
    throws IOException {

```

```

InputStream fis = null;
InputStreamReader isr = null;
BufferedReader bufferedReader;
try {
    fis = application.openFileInput(
        SWDApplication.getUserWasteCalendarFileName());
    isr = new InputStreamReader(fis, IO.UTF_8);
    bufferedReader = new BufferedReader(isr);
    final StringBuilder sb = new StringBuilder(256);
    String line;
    while ((line = bufferedReader.readLine()) != null) {
        sb.append(line);
    }
    return gson.fromJson(sb.toString(), UserSettings.class);
} catch (Exception e) {
    Log.e(LOG_TAG, "Failed to load user address", e);
    return new UserSettings();
} finally {
    IO.close(fis);
}
}

public static void saveUserWasteAddress(@NotNull final UserSettings adress, @NotNull
    final SWDApplication applicationObject) throws IOException {
    FileOutputStream outputStream = null;
    try {
        final String json = gson.toJson(adress);
        outputStream = applicationObject.openFileOutput(
            SWDApplication.getUserWasteCalendarFileName(), Context
            .MODE_PRIVATE);
        outputStream.write(json.getBytes());
    } catch (Exception e) {
        IO.reportException(e);
        Log.d(LOG_TAG, "", e);
    }
}
}

```

B11: saveUserWasteAddress und read

```

public static void saveUserWasteAddress(@NotNull final UserSettings address,
    @NotNull final SWDApplication applicationObject) throws IOException {
    try {
        final String json = gson.toJson(address);
        final SecureAndroid secureAndroid = new SecureAndroid(
            applicationObject.getApplicationContext());
        secureAndroid.encryptAndStore(SecureAndroid.SHARED_PREFERENCES,
            json.getBytes("UTF-8"), SWDApplication.
            getUserWasteCalendarFileName());
    } catch (Exception e) {
        IO.reportException(e);
        Log.d(LOG_TAG, "saveUserWasteAddress", e);
    }
}

public static UserSettings read(@NotNull final SWDApplication application)
    throws IOException {
    try {
        final SecureAndroid secureAndroid = new SecureAndroid(
            application.getApplicationContext());
        return gson.fromJson(new String(secureAndroid.retrieve(
            SecureAndroid.SHARED_PREFERENCES,
            SWDApplication.getUserWasteCalendarFileName()), "UTF-8"),
            UserSettings.class);
    } catch (Exception e) {
        Log.e(LOG_TAG, "Failed to load user address", e);
        return new UserSettings();
    }
}

```

B12: Angepasste read-Methode

```
public static UserSettings read(@NonNull final SWSApplication application)
    throws IOException {
    try {
        SecureAndroid sc = new SecureAndroid(application, 1000);
        return gson.fromJson(new String(sc.retrieve(
            SecureAndroid.SHARED_PREFERENCES, SWSApplication.
            getUserWasteCalendarFileName()), "UTF-8"), UserSettings.class);
    } catch (DataNotAvailableException|NoKeyMaterialException e) {
        final UserSettings tempUserSettings = readOld(application);
        final File file = new File(application.getFilesDir(), SWSApplication.
            getUserWasteCalendarFileName());
        saveUserWasteAdress(tempUserSettings, application);
        file.delete();
        return read(application);
    } catch (Exception e) {
        Log.e(LOG_TAG, "Failed to load user address", e);
        return new UserSettings();
    }
}
```

Eidesstattliche Erklärung zur Bachelorarbeit

Ich versichere, die Bachelorarbeit selbstständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben.

Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Hattingen, den 16.02.2016

Daniel Sahm