Bachelorarbeit

Detektion von Shared Preferences-Einträgen in Android-Applikationen mit Hilfe statischer Programmanalyse

Maximilian Schönborn
Matrikelnr. 2044977

21.03.2016

Universität Bremen

Fachbereich 3: Mathematik und Informatik
Studiengang Informatik (Vollfach)

Erstgutachter: Dr. Karsten Sohr
Zweitgutachterin: Prof. Dr. Ute Bormann

Urheberrechtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die ich wörtlich oder sinngemäß aus anderen Werken entnommen habe, habe ich unter Angabe der Quellen als solche kenntlich gemacht.

Datum Unterschrift

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, deren Unterstützung die Realisation dieser Abschlussarbeit in der vorliegenden Form ermöglicht hat:

Mein Dank gilt zunächst Dr. Karsten Sohr, welcher mich seit der Themenfindung als Betreuer durch die Ausarbeitung dieser Arbeit begleitet hat. Seine Erfahrung und wertvollen Ratschläge haben mir sehr geholfen.

Auch danke ich Bernhard Berger, verantwortlich für Konzeption und Entwicklung des ZertApps-Demonstrators, für die umfangreiche technische Hilfestellung im Umgang mit Soot.

Bei meinen Freundinnen Eileen und Katharina bedanke ich mich insbesonders für das Einbringen vieler sprachlicher und formaler Verbesserungs- und Korrekturvorschläge.

Ein besonders herzlicher Dank gilt meinen Eltern, die mir mein Studium ermöglicht und mich stets ermutigt haben.

Inhaltsverzeichnis

1 Einleitung

Smart Devices, insbesondere Mobiltelefone, sind in privater oder professioneller Nutzung für viele Menschen ständige Begleiter im Alltag geworden. Das Hinzufügen von Funktionen durch Installation von Software, in diesem Kontext und im folgenden Apps (geläufige Kurzform für Applications oder Applikationen) genannt, ist ein wesentliches Herausstellungsmerkmal dieser Kategorie von Geräten. Je nach Anwendungszweck verwaltet eine App mitunter sensible Informationen wie beispielsweise Zugangsdaten zu anderen Diensten und Systemen oder gar persönliche Daten. Dass jene besonders sorgsam behandelt werden, müssen steht zwar außer Frage, allerdings liegt die Vermutung nahe, dass dies möglicherweise nicht immer der Fall ist. Gründe dafür könnten unter anderem Fehlannahmen hinsichtlich der Sensibilität, mangelnde Domänenkenntnisse, mangelndes technisches Verständis der Plattform oder gar Nachlässigkeit seitens der Softwareentwickler sein.

Zu den bedeutendsten Plattformen für mobile Anwendungen gehört Googles Betriebssystem Android. Über den Google Play Store und weitere Kanäle stehen dem Benutzer dort eine mittlerweile praktisch unüberschaubare Auswahl von mehr als 1,8 Mio. Apps zur Verfügung [37]. Aufgrund seiner große Verbreitung, dazu mehr in Abschnitt 2.1, stellt Android ein interessantes Ziel für potentielle Angriffe dar. Unter diesem Gesichtspunkt sind von den Apps auf dem Gerät verwaltete Daten besonders interessant. Eine App beziehungsweise deren Entwickler kann unter Android zwischen mehreren Möglichkeiten der persistenten Datenspeicherung wählen. Eine davon sind die SharedPreferences, welche in Form von unverschlüsselten Einträgen in Extensible Markup Language (XML) Dateien innerhalb des Dateisystems vorliegen. Im schlechtesten Fall können sie sogar für alle lokalen Nutzer lesbar angelegt werden. In ihnen gespeicherte Informationen sind potentiell gefährdet, da in den letzten Jahren immer wieder Sicherheitslücken bekannt wurden, welche es Angreifern ermöglichten, nicht vorgesehene Rechte und Privilegien zu erlangen. Die 2015 durch Zimperium, Inc. [43] veröffentlichte, wahrscheinlich bisher bedeutendste davon nutzt die Stagefright-Multimediaschnittstelle aus und war über den Erhalt von, mit Exploit-Code versehenen, Multimedia Messaging Service (MMS)-Nachrichten auslösbar. Das Android-Website monatlich ausgegebene Nexus Security Bulletin gibt allein für

den Zeitraum November 2015 bis Januar 2016 elf neue Verwundbarkeiten (Vulnerabilities) der höchsten Dringlichkeitsstufe bekannt [9][7][8]. Als Auswirkung davon können Android-interne Schutzmechanismen übergangen werden und somit möglicherweise unberechtiger Zugriff auf eigentlich geschützte Dateien erlangt werden. Besonders brisant wäre daher eine Speicherung von Zugangsdaten jeglicher Art in Form von SharedPreferences-Einträgen. Dies würde beispielsweise bei Kommunikationsapps für Email oder Instant Messaging im schlimmsten Fall eine Kontoübernahme im Sinne eines Identitätsdiebstahls ermöglichen. Ebenfalls vorstellbar sind Risiken in Verbindung mit per App steuerbaren onlinefähigen Haushaltsgeräten, Bestandteilen von Smart-Home Systemen oder anderen Manifestationen des Internet der Dinge.

Verfahren der statischen Programmanalyse bieten die Möglichkeit, Aspekte des Daten- und Kontrollflusses von Software zu untersuchen ohne, diese auszuführen, und lassen sich damit zur Bewertung der aufgeführten Annahme nutzen. Bezogen auf die Plattform Android und einem bestimmten Typ von Datenspeicher, den SharedPreferences, wird im Rahmen der vorliegenden Arbeit ein Verfahren dazu entwickelt. Dieses geschieht im Kontext des Forschungsprojekt ZertApps, welches eine Zertifizierung von Apps unter sicherheitsrelevanten Gesichtspunkten anstrebt und in Abschnitt 2.3.2 vorgestellt wird. Zur Evaluation wird das Verfahren an einer Benchmarking-Suite getestet und zur Durchführung einer stichprobenhaften Fallstudie auf einer Auswahl von Apps verwendet. Das als Implementation des Verfahrens geschaffene Werkzeug soll nach Eingabe einer App in Binärform, als Android Package (APK)-Datei, je innerhalb ihrer Komponenten und deren Methoden lokal Aufrufe erkennen. Anhand deren Parameter lassen sich Name und Berechtigungen der SharedPeferences-Dateien und -Schlüssel sowie zugehörige Datentypen in den jeweils aufgerufenen Einträgen feststellen. Unter Berücksichtigung der vorherigen Evaluation des Verfahrens und ergänzenden manuellen Begutachtung von Code-Fragmenten soll das Ergebnis der Fallstudie eine fundierte Ersteinschätzung des Risikos der möglicherweise unbedarften Verwendung der persistenten Speicherung von Daten in SharedPreferences-Dateien bieten.

Aktuelle Veröffentlichungen der letzten drei Jahre, welche SharedPreferences in Android-Apps explizit in einem Sicherheitskontext thematisieren konnten nur wenige ermittelt werden. Reaves u. a. [35] untersuchte sieben ausgewählte Apps aus dem Finanzsektor, wobei eines der untersuchten Kriterien die Verwendung von SharedPreferences ist. Es wurden dazu keine speziellen Werkzeuge verwendet, sonder eine manuelle Inspektion von dekompilierten Quelltexten durchgeführt. Drei der Apps nutzten diese für das Speichern von sensibles Informationen, etwa persönliche Daten oder auch der Personal Identification

Number (PIN). Li u. a. [31] stellen mit IccTA ein statisches Analysetool zur Feststellung von Datenlecks auch über mehrere Komponenten (inter-component) von Android-Apps vor. Im Zuge der Evaluierung konnten auf einer Auswahl von unter anderem mehr als 15.000 zufällig aus dem Google Play Store bezogenen Apps die SharedPreferences als häufigster Senke (Ziel eines Datenflusses) ermittelt werden.

2 Grundlagen

In diesem Kapitel werden die für das Verständnis dieser Arbeit erforderlichen inhaltlichen Grundlagen zu Android, statischer Programmanalyse und dem Kontext der Implementierung zusammengefasst. Es ist allerdings anzumerken, dass ein Basiswissen im Bereich von Betriebssystemen und der Programmiersprache Java vorausgesetzt wird. Da dessen Aufbereitung den inhaltlichen Rahmen sprengen würde, sei dafür beispielhaft auf Tanenbaum und Bos [38] und die Java Dokumentation [33] verwiesen. Das Betriebssystem Android wird zunächst vorgestellt und im Weiteren dann auf seine Systemarchitektur, sein Sicherheitsmodell und die SharedPreferences eingegangen. Da verwendete Literatur teilweise keine aktuelleren Angaben enthielt, beschränkt sich die Gültigkeit der Ausführungen auf Versionen bis einschließlich 5.1 Lollipop. Anschließend werden die statische Programmanalyse und darauf aufbauend das Soot-Framework und das ZertApps-Projekt, mit den daraus verwendeten Werkzeugen, vorgestellt.

2.1 Android

Android wurde ursprünglich als Betriebssystem für Mobilgeräte mit Touchscreen, wie Smartphones und später Tablets, konzipiert. Basierend auf quelloffener, freier Software wird es nach Übernahme von Android, Inc. federführend von Google entwickelt. Gefördert und vertreten wird Android als offene Plattform durch die Open Handset Alliance, ein Konsortium von Netzbetreibern, Geräte-, Hardware-, und Softwareherstellern.

Aktuell diversifiziert sich die Ausrichtung Androids zunehmend in neue Anwendungsgebiete und Gerätekategorien, wie etwa Wearables, TV Set-Top-Boxen oder Bordcomputer in Automobilen. In seinem ursprünglichen und wichtigsten Markt ist Android inzwischen global gesehen führend. Schätzungen von Marktforschungsfirmen zufolge wurden mehr als 80% der in 2014 verkauften Smartphones (nach Gartner Inc. [21] 80,7% von 1244,9 Millionen, laut Mawston [32] 81,2% von 1283,5 Mio. und gemäß International Data Corporation (IDC) [28] 81,5% von 1300,4 Mio.) mit Googles Betriebssystem ausgeliefert.

2.1.1 Systemarchitektur

Die Architektur des Android-Betriebssystems ist, wie aus Abbildung 2.1 ersichtlich, im Wesentlichen ein Schichtenmodell. Dessen Beschreibung und Vorstellung seiner Bestandteile in diesem Abschnitt basiert auf den einführenden Kapiteln von Elenkov [20, Seite 3 bis 7] und Zhauniarovich [42, Seite 1 bis 12]. Bei Betrachtung in aufsteigender Abstraktion, von der Hardware zur Anwender-Software, bildet der Linux Kernel die Grundlage. Die darauf aufsetzenden Komponenten Init, Native Daemons, Native Libraries und Hardware Abstraction Layer (HAL) bilden den Native Userspace, die zweite Schicht. Auf die folgende, als Dalvik Runtime bezeichnete, Dalvik Virtual Machine (VM) setzt das Framework als vierte Schicht auf. Dessen Bestandteile sind die Android Framework Libraries, die System Services sowie die Java Runtime Libraries. Die letzte Schicht beinhaltet schließlich die Benutzer- und System-Apps.

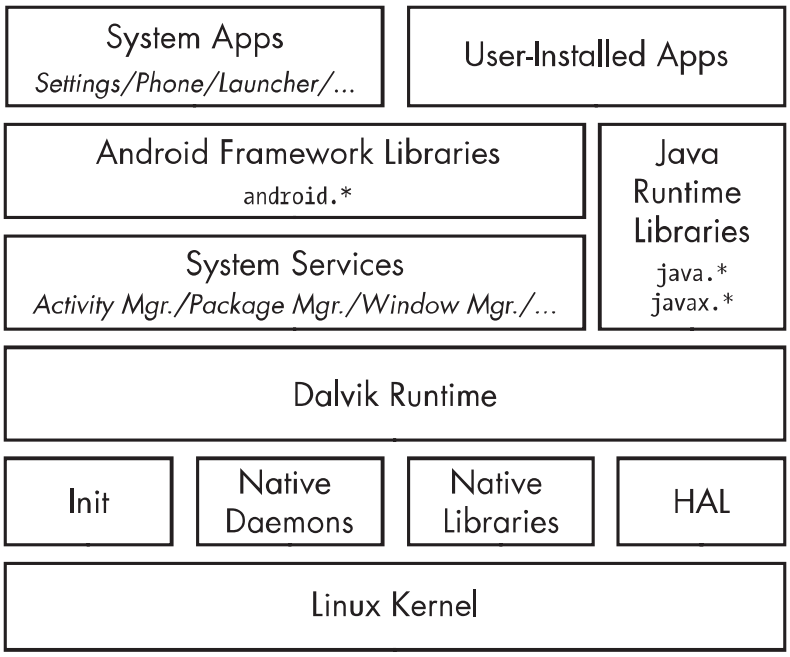


Abbildung 2.1: Android Systemarchitektur nach Elenkov [20]

Linux Kernel Bei dem in Android verwendeten Kernel handelt es sich einen modifizierten Linux Kernel. Wie bei unixoiden Betriebssystemen üblich, verwaltet er Prozesse, Speicher, Kommunikation, den Zugriff auf Dateisysteme und weitere Systemfunktionen über enthaltene Treiber. Wichtige plattformspezifische Ergänzungen beinhalten unter

anderem die Ashmem und Pmem Mechanismen zur Speicherverwaltung, Paranoid Networking zur Beschränkung von Netzzugriffen auf einzelne Prozesse, Binder zur Kommunkation zwischen Prozessen und Wakelocks für die Energieverwaltung.

Native Userspace Im Native Userspace befindliche Komponenten werden nativ, also außerhalb der, im folgenden Abschnitt vorgestellten, Dalvik VM, ausgeführt. Bestandteile sind HAL und Init, sowie die nativen Daemons und Libraries. Der HAL definiert standardisierte Application Programming Interfaces (APIs) zu den von Geräteherstellern zu implementierenden Treibern und abstrahiert so den Zugriff auf die Hardware. Dadurch ist es nicht mehr nötig, sämtliche Treibermodule in den Kernel einzubetten. Weiterhin ermöglicht die Verwendung von HAL die Deaktivierung des dynamischen Nachladens solcher Module. Der Init Prozess, welcher zum Abschluss des Kernelbootvorgangs aufgerufen wird, übernimmt wiederum das Starten sämtlicher weiterer Prozesse und Dienste. Zu der anschließenden Durchführung des Userspace Boots beinhaltet er zudem noch einige für das Betriebssystem wichtige Funktionen, wie beispielsweise das Überwachen und eventuelle Neustarten kritischer Prozesse.

Dalvik VM und Android Runtime (ART) Die Dalvik VM ist die Android-eigene virtuelle Maschine, welche das Ausführen der auf Java-Basis entwickelten Anwendungen ermöglicht. Im Hinblick auf die Ressourcen des Hostsystems und die Anforderungen der Gastsoftware entwickelt, weicht sie erheblich von der Oracle Java Virtual Machine (JVM) ab und kann nativen Java Bytecode (üblicherweise in Form von .class Dateien vorliegend) nicht ausführen. Stattdessen verarbeitet sie in .dex Dateien enthaltene Dalvik Executables (DEXs) mit dem eigenen Befehlssatz. Hauptunterschied zwischen der Dalvik VM und der JVM ist die Verwendung einer register-basierten Architektur bei ersterer, wohingegen die der letzteren stack-basiert ist. Daraus ergibt sich gemäß Bornstein [18, Folie 37] zusammen mit weiteren Optimierungen eine Einsparung von 30% der auszuführenden Instruktionen und damit ein deutlich effektiveres, wenngleich weniger universelles System. Seit dem 2014 erschienenen Android 5.0 Lollipop ersetzt ART, beispielsweise auf dem Android Entwicklerportal [2] nachvollziehbar, die damit obsolete Dalvik VM als Laufzeitumgebung. ART ist weitestgehend kompatibel: Sie übernimmt die DEX Bytecode Spezifikationen und führt die gleichen Dateien aus. Wichtigste Neuerung ist neben nativer 64-bit Unterstützung und einer verbesserten Garbage Collection vor allem die Einführung von Ahead-of-time Kompilierung. Wurde bei der Dalvik VM noch bei jedem Ausführen der Bytecode Just-in-time in Binärdateien übersetzt, erfolgt dies nun nur noch

einmal im Zuge der Ersteinrichtung einer App.

Framework Als Verbindung zwischen Laufzeitumgebung und Apps lassen sich Laufzeitbibliotheken (Java Runtime Libraries), Systemdienste (System Services) und Android Framework Libraries verstehen. Die Laufzeitbibliotheken Androids sind ursprünglich dem Apache Harmony Projekt entliehen und wurden im Laufe der Zeit bis zum kompletten Austausch einzelner Komponenten angepasst. Ergänzt werden sie durch die Android Framework-Bibliotheken. Neben den Bausteinen für Android-Apps, wie den Basisklassen für Activities, Services und Content Providern sowie Graphical User Interface (GUI)-Elementen, enthalten diese Klassen zum Zugriff auf Hardwarefunktionen und Systemdienste. Letztere werden dabei immer über eine eigene Manager-Klasse angesprochen, beispielsweise sei der LocationManager für die Standortdienste erwähnt. Die dafür angelegten öffentlichen APIs werden über die Android Systemdienste, zur Verwendung durch Apps, bereitgestellt. Dazu kommunizieren selbige per Java Native Interface (JNI) mit dem unterliegenden Native Userspace.

Apps Apps stellen dem Nutzer einen Großteil der Funktionalität durch Interaktion mit dem System zur Verfügung. Die Definition der Grundausstattung durch Auswahl geeigneter System-Apps obliegt den Geräteherstellern und kann dementsprechend variieren. Über die weitere Installation von Apps lassen sich Funktionen erweitern oder hinzufügen. System-Apps sind Teile der Vorinstallation und üblicherweise über ein Unterverzeichnis von /system eingebunden. Da Nutzer dort keine Schreibrechte haben, gelten sie als sicherer als vom Benutzer installierte Apps und genießen höhere Privilegien. Mit Ausnahme von Aktualisierungen bestehen im Normalfall keine Möglichkeiten zur Änderung. Allerdings steht es dem Nutzer frei, für grundlegende Systembestandteile wie die Tastatur oder Sprachein- und ausgabe alternative Apps nachzuinstallieren. Neben Grundfunktionen zum Beispiel für Telefonie, Short Message Service (SMS) und zur Verwaltung von Kontakten können System-Apps auch vorinstallierte Nutzer-Apps sein. Nutzer-Apps werden in ein Verzeichnis der dedizierten Partition für Nutzerdaten, üblicherweise /data, installiert. Apps sind untereinander strikt getrennt und haben nur Zugriff auf Ressourcen, welche der Nutzer explizit bei der Installation freigegeben hat. Auf diese Sicherheitsmechanismen wird in Abschnitt 2.1.2 noch detailliert eingegangen. Android-Apps bestehen im Gegensatz zu einer konventionellen Anwendung, etwa im Desktop- oder Serverbereich, aus eher lose verbundenen Komponenten. Dieses Konzept fördert die Interaktion von Apps untereinander, welche dazu verschiedene Einstiegspunkte besitzen können. Erreicht werden

diese über Nutzerinteraktionen innerhalb der App, durch Aufrufe aus anderen Apps heraus oder von Systemereignissen aus. Die Auslieferung der Apps erfolgt als APK in Form signierter .apk Dateien. Darin gebündelt enthalten sind die ausführbaren DEX-Dateien, Ressourcen, Bibliotheken und ein sogenanntes Manifest. Die Datei AndroidManifest.xml definiert eine Reihe von Eigenschaften der App, unter anderem, analog zu Java, einen Paketnamen. Ebenfalls findet sich hierin eine Liste der Komponenten mit den zugehörigen Einstiegspunkten. Es werden die folgenden Typen von Komponenten unterschieden:

Activities sind Kombinationen von GUI-Elementen mit unterliegender Funktionalität, die sich dem Nutzer auf dem Bildschirm darstellen, vergleichbar etwa mit Fenstern einer Desktop Anwendung.

Services werden im Hintergrund und ohne GUI ausgeführt und eignen sich daher besonders für länger anhaltende und dauerhafte Aufgaben.

Broadcast Receiver reagieren auf systemweite Ereignisse, welche vom System oder von Nutzer-Apps ausgehen können.

Content Provider ermöglichen es einer App, für eine gemeinsame Nutzung mit anderen Apps, Schnittstellen zu ihrem Datenbestand anzubieten.

2.1.2 Sicherheitsmodell

Das Android Sicherheitsmodell zeichnet sich durch einige Kernkonzepte aus, welche hier vorgestellt werden sollen. Dabei dienen Elenkov [20, Seite 12 bis 15] und die Website des Android Open Source Project [14] [4] als Grundlagen der folgenden Übersicht. Auf Ebene des Kernels und Dateisystems findet eine, auf bestehenden Portable Operating System Interface (POSIX) Standards basierende, Separation von Privilegien statt. Für den Zugriff auf Ressourcen des Systems müssen Apps explizit Berechtigungen vom Nutzer erteilt bekommen. Die Interoperabilität von Prozessen, sowohl von Systembestandteilen als auch von Apps, wird schließlich über Binder reguliert.

Sandboxing Jeder App steht eine eigene Laufzeitumgebung zur Verfügung, die sich durch ihre Eigenschaften als sogenannte Sandbox ausweist. Im Wesentlichen bedeutet dies die Isolation von Apps untereinander und von Systembestandteilen. Für die Umsetzung wird auf die im Linux Kernel bereits vorhandene Discretionary Access Control (DAC), die identitätsbezogenen Zugriffskontrolle, aufgebaut. Zum Zeitpunkt der Installation einer App werden zunächst eine Benutzer- und Gruppenkennzeichnung zugeteilt. Diese

von POSIX bekannten User Identifier (UID)- und Group Identifier (GID)-Attribute werden dabei so errechnet, dass sie einzigartig sind, und bleiben unverändert. Im Dateisystem werden sie, in Verbindung mit entsprechenden Berechtigungen, als Eigentümer der Ordner für Appsdaten gesetzt. Es wird demnach jede Android-App stets unter einem eigenen Linux-Nutzer ausgeführt. Dies zwingt den Kernel zur Zuteilung von isolierten Speicherbereichen und einer möglichst ausgewogenen Priorisierung bezüglich Systemressourcen, wie beispielsweise Rechenzeit, Speicher oder Netzwerkbandbreite.

Permissions Zur Verwendung von bestimmten, geschützten Ressourcen und Diensten außerhalb ihrer eigenen Sandbox können sich Apps unterschiedliche Berechtigungen vom Benutzer einholen. Dabei kann es sich zum Beispiel um den Zugriff auf Mediendaten, eine mögliche Kamera, SMS oder Anrufdaten handeln. Die Freigabe dieser muss zum Zeitpunkt der Installation über einen Dialog für alle angeforderten Berechtigungen zusammen erteilt werden. Andernfalls ist keine Installation möglich.

Binder Die Kommunikation zwischen Prozessen, kurz Inter Process Communication (IPC), realisiert das Android-spezifische Binder-Framework mit einem eigenen Kernel-Treiber. Nach dem Prinzip eines Client-Server-Modells kann ein Prozess einer App darüber Methoden eines anderen Prozesses wie lokale Methoden aufrufen (Remote Procedure Call (RPC)). Abgesichert werden Binder-Aufrufe über die Kennzeichnung per UID/Process Identifier (PID) des aufrufenden Prozesses und anschließendem Abgleich dieser Informationen mit den Berechtigungen durch den angefragten Prozess.

2.1.3 SharedPreferences

Bei der Umsetzung von Android-Apps können Enwickler auf verschiedene Mechanismen zur persistenten Speicherung von Daten zurückgreifen [13]. Neben den zu untersuchenden SharedPreferences ermöglichen jene dies auch direkt im Dateisystem, in SQLite Datenbanken und über das Netz. Trotz ihres Namens sind die SharedPreferences nicht ausschließlich als Speicher für Präferenzen, hier im Sinne von Optionen oder Einstellungen, des Benutzers vorgesehen. Die Klasse bzw. deren Objekte dienen vielmehr als Schnittstelle zu einem vielseitigen Schlüssel-Wert Datenspeicher [10]. Die Werte, in Form der primitiven Datentypen Boolean, Float, Int, Long, String und StringSet, lassen sich jeweils über eine zugehörige Zeichenkette als Schlüssel erreichen. Für jeden Speicher gibt es dabei genau eine Instanz der Klasse (Singleton Entwurfsmuster), um Integrität und

Konsistenz sicherzustellen. Die Verwendung lässt sich am besten durch ein kurzes Beispiel (Listing 2.1) veranschaulichen.

```
1  // read
2  SharedPreferences preferences = getSharedPreferences("PrefsFile",
   ↪  MODE_PRIVATE);
3  boolean notTrue = preferences.getBoolean("notTrue", false);

4  // write
5  SharedPreferences.Editor editor = preferences.edit();
6  editor.putBoolean("notTrue", false);
7  editor.commit();
```

Listing 2.1: Beispiel zur Verwendung von SharedPreferences

Der Aufruf eines `SharedPreferences` Objektes in Zeile 2 ermöglicht dort zunächst nur lesende Abfragen wie jene in der darauf folgenden Zeile. Dieses Laden von Werten realisieren datentypspezifische `get`-Methoden. Zur Bearbeitung durch Speichern von Werten mittels dazu analoger `put`-Methoden wie ab Zeile 4 bedarf es zunächst eines `SharedP` ⌋ `references.Editor`. Der abschließende Aufruf von `commit()` überträgt die zuvor vorgenommenen Änderungen. Erwartungsgemäß gibt es ergänzend noch je einen Befehl zum Entfernen von einzelnen oder allen Einträgen, `remove(String key)` und `clear()`. Eine umfassende Beschreibung der SharedPreferences ist in der Dokumentation der Android-API [11] [12] zu finden. Die Speicherung der so verwalteten Daten erfolgt in einem geschützten Unterverzeichnis der jeweiligen Android-App als XML-Dateien. Der aus obigen Befehlen erzeugte Eintrag würde sich wie in Listing 2.2 ausgegeben lesen.

```
1  <?xml version='1.0' encoding='utf-8' standalone='yes' ?>
2  <map>
3  <boolean name="notTrue" value="false" />
4  </map>
```

Listing 2.2: SharedPreferences XML-Dateiinhalt zu Listing 2.1

2.2 Statische Programmanalyse

Eine Programmanalyse dient dem Gewinn von Informationen über das Verhalten eines ganzen Programmes oder einzelner Bestandteile. Typischerweise findet sie zur Optimierung oder Verifikation des Analysegegenstandes statt. Dieser Abschnitt stellt die statische Programmanalyse ihrem dynamischen Pendant gegenüber und in einer strukturellen Übersicht dar.

2.2.1 Abgrenzung zur dynamischen Programmanalyse

Wie schon aus den Bezeichnungen ableitbar, besteht der grundlegende Unterschied zwischen dynamischer und statischer Analyse in der Ausführung des zu untersuchenden Programmes. Die dynamische Analyse als Beobachtung des Laufzeitverhaltens mit notwendigen konkreten Eingaben setzt einen ausführbaren Prüfling und geeignete Eingabewerte voraus [29, Folie 140]. Sind diese Anforderungen erfüllt, lassen sich mit Bezug auf einzelne Programmteile und Anwendungsszenarien detaillierte Informationen erheben. Die statische Analyse findet anhand einer Beschreibung, meist in Form des Quelltextes, statt [29, Folie 158]. Da sie keine konkreten Eingaben erfordert, lassen sich aus ihr allgemeingültige Aussagen über den Prüfling treffen. Als nachteilig ist jedoch der durch die Komplexität bedingte höhere Aufwand zu nennen. Praktisch ist dadurch oft eine Abwägung zwischen Ressourcenbedarf und Genauigkeit der Analyseergebnisse erforderlich [29, Folie 161].

2.2.2 Ebenen der statischen Programmanalyse

Neben der Verwendung im Software-Reengineering als Teil eines Analysators bildet die statische Analyse ursprünglich die Basis von Compilern. Es sind für das Reengineering laut Koschke [29, ab Folie 47] folgende (Teil-) Analysen besonders hervorzuheben. Ergänzende Beschreibungen sind Aho [1] entnommen.

Lexikalische Analyse

Die lexikalische Analyse [1, Seite 5 bis 7, ausführlich ab Seite 109] und deren zugehörige Analysator-Komponente, der Lexer, bereiten ein als Quelltext vorliegendes Programm für die syntaktische Analyse vor. Ziel ist das Erzeugen des Tokenstroms, einer Folge von Token, aus der Eingabe. Token sind hier kleinste bedeutsame Einheiten einer Pro-

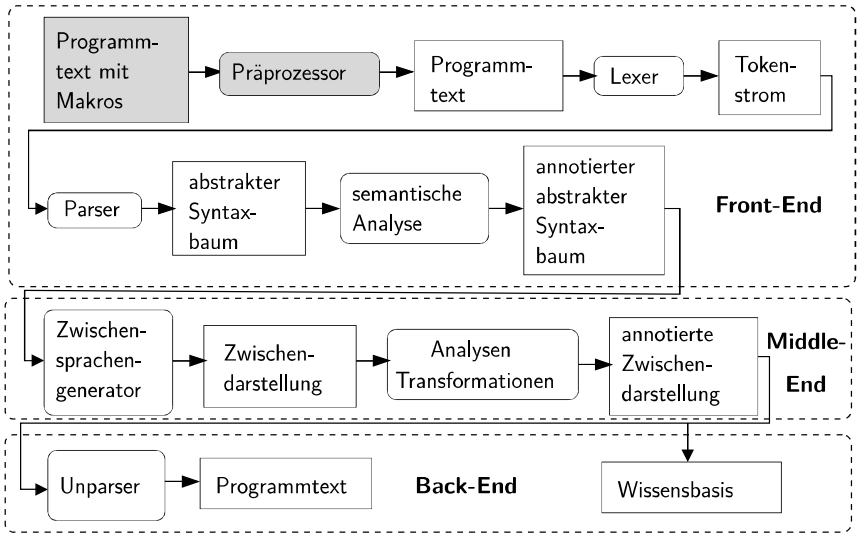


Abbildung 2.2: Analysator- und Transformator-Struktur (aus [29])

grammiersprache und umfassen Schlüsselwörter, Bezeichner, Literale, Operatoren und Trennzeichen, jedoch keine Whitespaces und Kommentare. Dazu erfasst der Lexer sequentiell im Quelltext vorkommende Lexeme, Zeichenketten, welche er anhand eines Musters (Pattern) als Instanz eines Token identifiziert. Die Muster sind üblicherweise über reguläre Ausdrücke definiert. Jedes erkannte Token wird, versehen mit Attributen wie Typ, Position im Quelltext sowie seinem Wert, dem Tokenstrom hinzugefügt.

Syntaktische Analyse

Darauf aufbauend findet die syntaktische Analyse, auch Parsing genannt, statt [1, Seite 8 bis 9, ausführlich ab Seite 191]. Die durchführende Analysator-Komponente heißt Parser und überprüft den vom Lexer kommenden Tokenstrom auf syntaktische Korrektheit im Sinne der verwendeten Programmiersprache. Außerdem erzeugt er im Erfolgsfall mit dem Syntaxbaum eine Repräsentation der Struktur des eingegebenen Programmes. Die als Prüfungsgrundlage dienenden Regeln liegen üblicherweise in kontextfreien Grammatiken vor. Bei den enthaltenen Symbolen werden Nichtterminale, durch Anwendung mindestens einer weiteren Regel ersetzbar, von Terminalen unterschieden. Die Ersetzungsregeln folgen stets der Backus-Naur-Form:

Nichtterminal - (Menge an Terminalen und Nichtterminalen)

Die Kontextfreiheit bezieht sich auf das jeweils alleinstehende Nichtterminal auf der linken Seite der Regeln. Damit sind Anwendungen stets unabhängig vom Kontext, etwaig vor- und nachstehenden Symbole, des zu ersetzenden Nichtterminals. Lässt sich die Eingabe durch Anwendung von Grammatikregeln herleiten, ergibt sich aus diesen Ersetzungen unmittelbar ein Ableitungs- oder konkreter Syntaxbaum. Dort sind Nichtterminale den Knoten und Terminale oder das leere Wort den Blättern zugeordnet. Parser werden grundlegend nach der Richtung ihres Vorgehens unterteilt. Die Konstruktion durch Erstellen der Knoten kann Top-Down, beginnend mit dem Startsymbol an der Wurzel absteigend, erfolgen. Oder umgekehrt Bottom-Up, beginnend mit den Terminalen als Blätter aufsteigend. Im Falle mehrdeutiger Grammatiken können durchaus mehrere verschiedene gültige Ableitungsbäume existieren. Durch Auslassung zur Erkennung notwendiger, jedoch für die weitere Verarbeitung irrelevanter, syntaktischer Detailinformationen lässt sich ein Abstract Syntax Tree (AST), also abstrakter Syntaxbaum, generieren. Je nach Anwendung variiert der Umfang dieser Abstraktionen. Der AST dient als Zwischenrepräsentation und Grundlage weiterer Analysen und Transformationen und kann im Zuge dieser mit weiteren Informationen versehen werden.

Kontrollflussanalyse

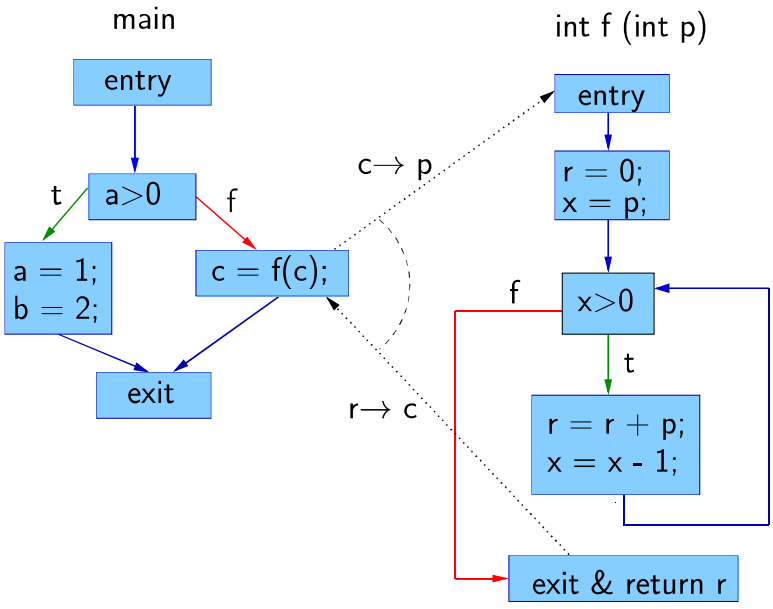


Abbildung 2.3: Intra- und interprozeduraler Kontrollfluss (aus [29])

Die Kontrollflussanalyse [1, ab Seite 525] dient der Untersuchung der möglichen Ausführungsreihenfolge und Erreichbarkeit von Anweisungen eines Programmes. Von Interesse sind insbesondere daraus entstehende Abhängigkeiten zwischen Teilen des Programmes. Beschrieben werden diese durch den Control Flow Graph (CFG). Die Knoten des CFG stellen, mit Ausnahme des Wurzelknotens als Einstiegspunkt und des Endknotens, je eine Anweisung oder einen verzweigunsfrei ausführbaren Anweisungsblock dar. Verbunden sind sie durch gerichtete Kanten, falls eine Erreichbarkeit untereinander existiert. Je nach Ziel der Analyse lassen sich der intraprozedurale Kontrollfluss innerhalb einer Methode, der interprozedurale Kontrollfluss zwischen Methoden oder eine Kombination aus beiden betrachten. Abbildung 2.3 zeigt als Beispiel für letzteres einen CFG für *main* und *f*. Ein- und Ausstiegsknoten sind mit *entry* respektive *exit* gekennzeichnet. Bei Vergleichsoperationen spaltet sich der Kontrollfluss, wobei grüne Kanten (t) den Fall des Zutreffens und rote Kanten (f) das Gegenteil repräsentieren. Gepunktete Kanten stellen interprozedurale Aufrufe mit Zuweisung der Variablen dar. Die Begriffe Dominanz, Postdominanz und, darauf aufbauend, Kontrollabhängigkeit beschreiben, welche Blöcke und damit Anweisungen zu einem gewählten Punkt des Programmes garantiert vorher und nachher ausgeführt werden. Dominanz eines Knotens A über B bedeutet, dass alle Wege vom Startknoten nach B stets über A führen. Postdominanz eines Knotens A über B bedeutet hingegen, dass alle Wege von B zum Endknoten stets über A führen.

Datenflussanalyse

Die Datenflussanalyse [1, ab Seite 597] beschäftigt sich, meist anhand des Kontrollflussgraphen, mit der Veränderung von Daten in und zwischen den Anweisungsblöcken und damit möglicherweise bestehenden Abhängigkeiten. Als relevante Operationen gelten ein Setzen (Set) oder Verwenden (Use) eines Wertes, welche unterschiedliche Beziehungen zwischen zwei im Kontrollfluss verbundenen Anweisungsblöcken 1 und 2 ergeben können. Wird ein Wert in 1 gesetzt und in 2 verwendet, ist dies eine Set-Use Beziehung oder Datenabhängigkeit. Wird ein Wert hingegen zuerst in 1 verwendet und in 2 überschrieben, wird dies als Use-Set-Beziehung oder Anti-Dependency bezeichnet. Bei einer Set-Set-Beziehung oder Output-Dependency wird wiederum ein in 1 gesetzter Wert in 2 überschrieben. Lokal werden pro Grundblock X, das ist ein nicht-verzweigter Abschnitt von Anweisungen im Kontrollfluss, die sogenannten GEN(X)- und KILL(X)-Mengen betrachtet. In GEN enthalten sind alle Zuweisungen innerhalb des Blocks, die dessen Ende erreichen. KILL beinhaltet alle lokalen Neuzuweisungen von Variablen, die nicht in GEN enthalten sind, somit an anderer Stelle des Kontrollfluss bereits zugewiesen waren. Für

die globale Datenflussanalyse sind zu jedem Grundblock X zusätzlich die Mengen in(X) und out(X) wichtig. Diese umfassen jeweils alle Definitionen, die den Eingang respektive den Ausgang von X erreichen. Je nach dem Ziel der Untersuchung kann diese mit oder gegen den Kontrollfluss erfolgen. Aho [1] erläutert als wichtige Probleme die Ermittlung von:

- **Reaching definitions** (Erreichende Definitionen, Vorwärtsproblem): Definitionen erreichen einen Punkt im Programmablauf, wenn ihrer Variable auf dem Weg zu diesem kein neuer Wert zugewiesen wird. [1, ab Seite 601]
- **Live variables** (Lebendige Variablen, Rückwärtsproblem): Variablen sind lebendig, wenn deren Werte an einem Punkt im Programmablauf vor der nächsten Zuweisung noch verwendet werden. Variablen sind zu Beginn eines Blocks lebendig, werden dort genutzt und danach neu zugewiesen, oder zum Ende eine Blocks, wenn sie innerhalb des Blocks nicht neu zugewiesen werden. [1, ab Seite 608]
- **Available expressions** (Verfügbare Ausdrücke, Vorwärtsproblem): Verfügbar ist ein Ausdruck zu einem Punkt im Programmablauf, wenn er auf jedem Pfad vom Eingangsknoten zu diesem Punkt berechnet wird und anschließend keine Neuzuweisung seiner Bestandteile ohne erneute Auswertung erfolgt. [1, ab Seite 610]

2.3 Umgebung

2.3.1 Soot-Framework

Soot ist ein Framework zur statischen Analyse von in Java programmierter Software. Es ist zur Erweiterung und als Grundlage zur Implementierung eigener Untersuchungswerkzeuge durch seine Nutzer vorgesehen. Ursprünglich an der Sable Research Group der McGill University entwickelt, wird es mittlerweile hauptsächlich durch die Forschungsgruppe Secure Software Engineering der TU Darmstadt gepflegt. Prinzipiell funktioniert Soot, wie bereits zuvor in Abschnitt 2.2.2 angedeutet, wie ein Compiler. Ausgehend von einer Eingabe, als Quelltext oder Bytecode (letzteres für Java oder Dalvik), erfolgt eine mehrstufige Verarbeitung in analysier- und transformierbaren Zwischenrepräsentationen, den Intermediate Representations (IRs). Ausgegeben werden kann, neben Informationen in verschiedenen Formaten, der modifizierte, ausführbare Bytecode, um beispielsweise eine Instrumentierung vorzunehmen. Im Zuge der Ausführung durchläuft Soot aufeinanderfolgend sogenannte Packs, welche sich wiederum in Phasen untergliedern. Zur Ver-

wendung ist vorgesehen, dass Nutzer Packs durch eigene Durchläufe ergänzen. Diese werden nach zwei Arten von Transformatoren unterschieden: Ein BodyTransformer ist für intraprozedurale Analysen konzipiert und wird auf jede Methode eines zu untersuchenden Programmes angewendet. Ein SceneTransformer wird hingegen einmal auf das gesamte zu untersuchende Programm angewendet und ist für interprozedurale Analysen geeignet. Insgesamt bietet Soot vier eigene IRs, in steigender Abstraktion; ausgehend von Java-Bytecode sind dies Baf, Jimple, Shimple und Grimp. Baf ist eine gestraffte (*streamlined*), besser manipulierbare IR des Bytecode. Jimple ist eine für Optimierungen besonders geeignete, typisierte Three Address Code (TAC)-Repräsentation. Instruktionen bestehen in einer TAC-IR aus maximal drei Operanden, wovon, nach dem Muster $a = b\ op\ c$, stets einer über eine Zuweisung Resultat einer Operation auf den anderen beiden ist. Shimple ist eine in der Static Single Assignment (SSA)-Form gehaltene Variation von Jimple. Eine SSA-Repräsentation unterscheidet sich von der TAC-Form dahingehend, dass jede Variable nur einmal zugewiesen werden kann und vor Verwendung bereits definiert sein muss. Grimp ist eine aggregierte Variante von Jimple, welche besonders für Dekompilierung und Code-Inspektion geeignet ist. Am bedeutendsten von diesen ist Jimple, welches Shimple und Grimp zu Grunde liegt. Im ersten Schritt eines jeden Soot-Laufes wird Jimple zunächst zur Repräsentation sämtlicher Methoden des zu untersuchenden Programmes und, darauf aufbauend, der meisten weiteren Analysen verwendet. Hinsichtlich seiner Untersuchbarkeit unterscheidet sich Jimple, neben der bereits genannten TAC-Form, vor allem durch das Speichern von Daten in lokalen, benannten Variablen anstatt des, in Java-Bytecode sonst üblichen, impliziten Stacks. Zudem zeichnet es sich durch einen stark reduzierten Satz von Operationen und Typsicherheit aus.

Beispielhafte Listings von Jimple-Instruktionen finden sich im Rahmen von Erkennung und Auswertung bestimmter Befehle im Abschnitt 3.2. Weitere Details zur Entwicklung und dem Funktionsumfang bis 2011 führen Lam u. a. [30] aus. Aktuelle Ergänzungen sind der Soot-Homepage [36] entnommen.

2.3.2 ZertApps-Projekt

Vor dem Hintergrund einer zunehmenden Verbreitung von Smart-Devices und des praktisch unüberschaubaren Angebotes an Apps, welche im Zusammenspiel erhöhte Risiken von Schadsoftware und Schwachstellen und damit potentiellen Informationslecks bergen, beschäftigt sich das Forschungsprojekt ZertApps mit der Entwicklung einer Analyse- und Zertifizierungsplattform für Android-Apps. Seine erstmalige Veröffentlichung durch Bartsch u. a. [16] dient, ergänzt durch aktuelle Informationen von der Internetpräsenz

[41], als Grundlage für diesen Abschnitt. Getragen wird das Projekt durch ein Konsortium von Partnern aus Wirtschaft und Forschung, gefördert vom Bundesministerium für Bildung und Forschung. Aufgrund seiner weiten Verbreitung wurde das Android Betriebssystem als Ausgangspunkt gewählt, mit der Perspektive, Ergebnisse zu einem späteren Zeitpunkt auf andere Systeme übertragen zu können. Angedacht ist, dass Entwickler von Apps diese über eine Prüfungsstelle testen lassen können, welche ihnen die Einhaltung von bestimmten Sicherheitsrichtlinien bestätigt. Die Resultate werden in Form von Zertifikaten zur Verfügung gestellt und können für Sicherheitseinstufungen automatisiert von Distributionsplatformen ausgewertet oder als Ausgangspunkt weiterer manueller Untersuchungen genutzt werden. Die Basis der Zertifizierung bildet ein statisches Analyseverfahren, welches einen, im Gegensatz zu bestehenden Werkzeugen, umfassenderen Ansatz verfolgt. Insbesondere beinhaltet dieser die Untersuchung der Kommunikation zwischen Prozessen unter Android-spezifischen Aspekten und die in *Summaries* zusammengefasste, komponentenweise Bewertung von Apps und Systembestandteilen. Die für den Nutzer interpretier- und nachvollziehbaren Endergebnisse sollen aus den Teilanalysen abgeleitet werden.

Prototyp Im Zeitraum der Fertigstellung dieser Arbeit entsteht am Technologie-Zentrum Informatik und Informationstechnik an der Universität Bremen dazu ein Demonstrator. Das zuvor vorgestellte Soot wurde durch dahingehend erweitert, dass es ein Eclipse Modelling Framework (EMF) Ecore-Domänenmodell instanziert und dieses mit über Plugins erzeugten Analyseergebnissen annotiert. Neben dem automatisierten Erstellen von Data Flow Diagrams (DFDs) [40, Mitteilung unter *Bremen, 27. Nov. 2014*] werden aktuell weitere Analysen und Funktionen integriert. Die Verwendung dieses Demonstrators ist mit einer als APK im Binärformat vorliegenden Android-App vorgesehen und erfordert somit keine Kenntnis des Quelltext. Steuerbar ist der Analyseprozess über das Verändern von Parametern des Aufrufs und Konfigurationsdateien in der Eclipse-Entwicklungsumgebung. Nachdem zunächst aus der Android-App ein Datenflussdiagramm erzeugt wird, erfolgt die Analyse dessen im Hinblick auf architekturelle Verwundbarkeiten. Das Untersuchungsergebnis in Form des annotierten DFD, bietet eine Übersicht über die Software-Architektur und mögliche enthaltene Sicherheitsrisiken. Da die Erweiterung des Demonstrators Gegenstand des praktischen Anteils dieser Arbeit ist, werden das Modell, im Zuge dessen Erweiterung in Abschnitt 3.1, und die Integration einer Analysekomponente zu dessen Anreicherung, in Abschnitt 3.2, noch teilweise vertieft.

3 Umsetzung

In diesem Kapitel wird die praktische Komponente dieser Arbeit vorgestellt. Dazu werden die vorausgehende Modellierung und die Implementierung als Softwaremodul für den ZertApps-Prototypen beschrieben.

3.1 Ergänzung des Modells

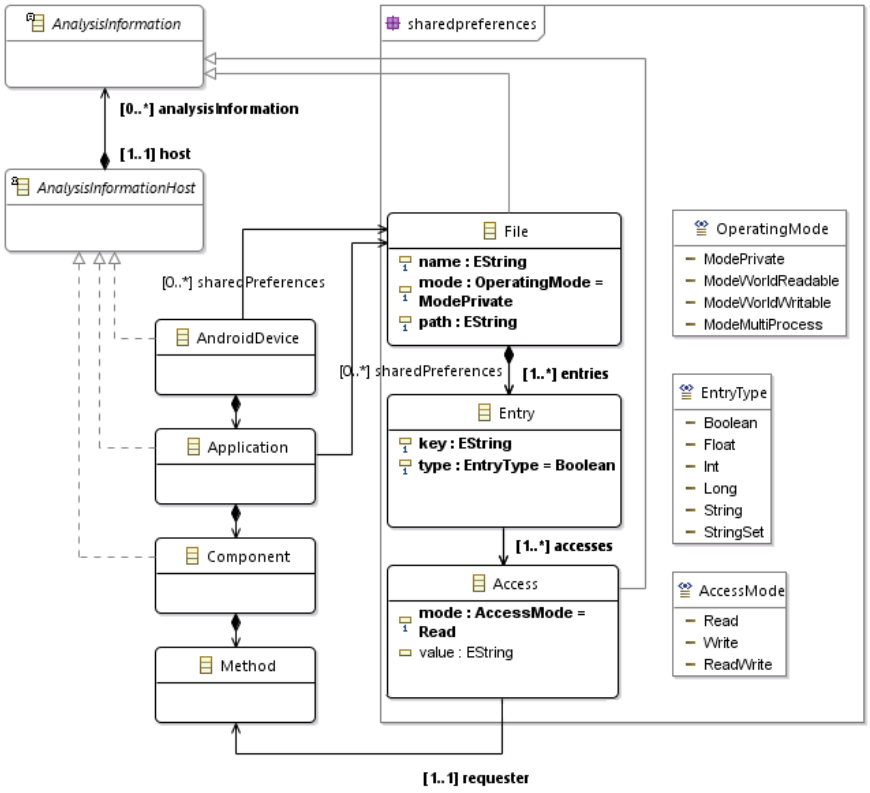


Abbildung 3.1: Ergänztes Ecore-Modell

Um die Analyseergebnisse im vorhandenen Ecore-Domänenmodell erfassen zu können, musste dieses zunächst erweitert werden. Da sich das EMF zur Beschreibung seiner Domänenmodelle geläufiger Unified Modeling Language (UML)-Konventionen bedient, kann das Diagramm in Abbildung 3.1 bereits eine Übersicht über die vorgenommenen Ergänzungen vermitteln. Die Darstellung ist dabei auf eine Teilmenge des Modells beschränkt, welche nur für diese Arbeit relevante Elemente beinhaltet. Präziser ausgedrückt sind darin nur jene Bestandteile des Gesamtmodells enthalten, welche mit den neu eingeführten Elementen in Beziehung stehen. Diese Neuerungen sind dem `sharedpreferences`-Namespace untergeordnet, welcher die rechte Hälfte des Diagramms bildet. Die dort angeordneten Klassen `File`, `Entry` und `Access` schließen sich an links davon befindliche, zuvor bereits existierenden Entitäten an. Diese beschreiben die Aggregationsbeziehungen zwischen einem Gerät (`AndroidDevice`), den darauf installierten Apps (`Application`), deren Komponenten (`Component`) und den wiederum in diesen beinhalteten Methoden (`Method`). Das abstrakte Interface `AnalysisInformationHost`, welches auf Geräte-, App- und Komponentenebene zur Verfügung steht, bietet über die abstrakte Klasse `AnalysisInformation` die Möglichkeit, Informationen mit Objekten zu verknüpfen. Die im `sharedpreferenc`⌋`es`-Namespace enthaltenen Klassen mitsamt Attributen sowie deren Einbindung in das bestehende Modell sind, in absteigender Reihenfolge:

File Die Klasse File repräsentiert eine SharedPreferences-Datei. Ihre Attribute sind Dateiname (`name`), Zugriffsberechtigung (`mode`) und Pfad im Dateisystem (`path`). Die Zugriffsberechtigung ist als Modus in Form der Enumeration `OperatingMode` umgesetzt und mit den möglichen Werten rechts der Klasse im Diagramm zu finden. Die Ausprägung dieser Eigenschaft bestimmt nachher die Zuordnung eines `File`-Objektes zu einem `AndroidDevice` oder einer `Application`. Bei einem Wert von `ModeWorldReadable` oder `M`⌋`odeWorldWriteable`, also einer Zugriffsmöglichkeit außerhalb der Applikation, wird eine Assoziation mit dem Gerät vorgenommen. Andernfalls wird, bei einer Sichtbarkeit innerhalb der App, die Datei mit dieser verknüpft. Jede Datei enthält ihrerseits Einträge, welche über die `entries`-Beziehung aggregiert sind.

Entry Die Klasse Entry repräsentiert einen Eintrag in einer SharedPreferences-Datei. Ihre Attribute sind Schlüssel (`key`) und Datentyp (`type`). Der Datentyp ist als Enumeration `EntryType` realisiert und mit den möglichen Werten ebenfalls rechts der Klasse im Diagramm zu finden. Da die Einträge über ihre Verwendung, also die Zugriffe, erkennbar sind, besteht eine `accesses`-Beziehung zu mindestens einem solchen.

Access Die Klasse Access repräsentiert einen Zugriff auf einen Eintrag einer Shared-Preferences-Datei. Ihre Attribute sind Zugriffsmodus (`mode`) und Wert (`value`). Der Zugriffsmodus ist als Enumeration `AccessMode` modelliert und mit den möglichen Werten genauso rechts der Klasse im Diagramm zu finden. Über die `requester`-Beziehung ist jedem Zugriff genau eine aufrufende Methode zugeordnet.

3.2 Implementierung

Die Vorstellung der implementierten Analyse gliedert sich entsprechend ihrer Struktur. Auf die Erläuterung der Vorgehensweise und des Algorithmus sowie einiger technischer Notwendigkeiten folgen die Beschreibungen der Java-Klassen sowohl von der Analyse als auch jene der zur Zwischenspeicherung und Verarbeitung der Ergebnisse verwendeten Objekte.

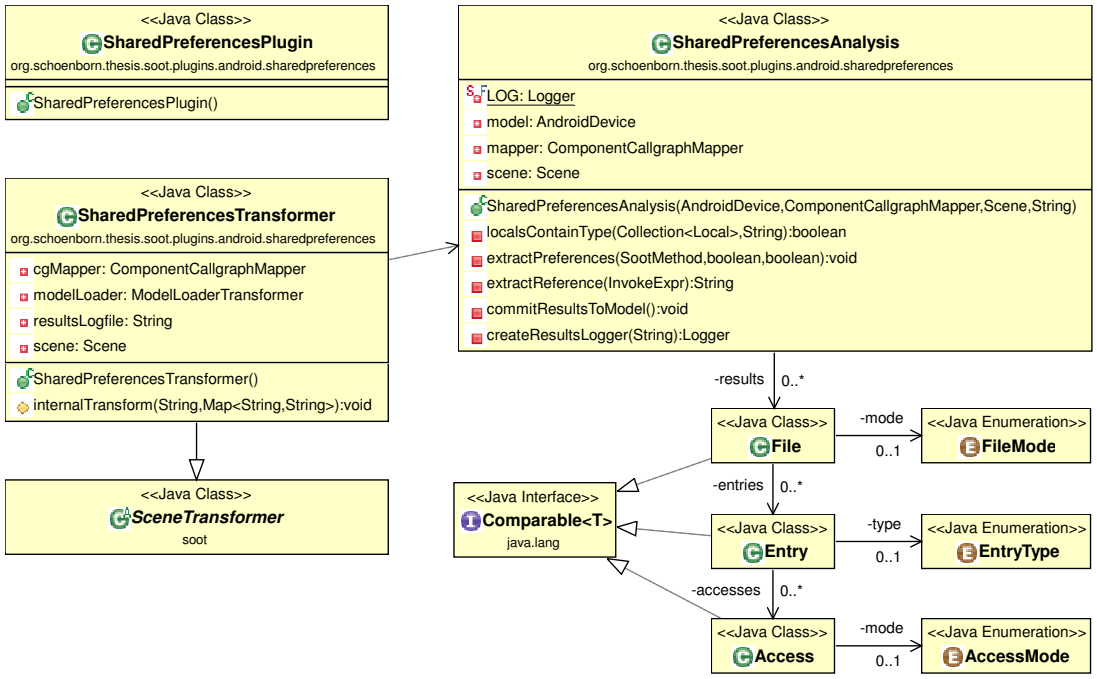


Abbildung 3.2: UML-Darstellung der Implementierung (Analyse-Klassen)

3.2.1 Übersicht

Bei der umgesetzten Untersuchung handelt es sich um eine lokale, intraprozedurale Programmanalyse auf Basis der Jimple-Zwischenrepräsentation einer App. Da die Analyse innerhalb der Klasse `SharedPreferencesTransformer`, welche von Soots `SceneTransformer` erbt, aufgerufen wird, kann sie auf alle in der Szene (Scene), welche zur internen Verwaltung sämtlicher Soot-Klassen und Soot-Methoden der zu untersuchenden App dient, enthaltenen Informationen zugreifen. Über alle Klassen und die in ihnen enthaltenen Methoden wird je festgestellt, ob es innerhalb einer Methode zur Verwendung von SharedPreferences-Objekten kommt um diese gegebenenfalls zu erfassen und zu protokollieren. Die Voruntersuchung erfolgt über das Auslesen der lokalen Variablen der aktuellen Methode. Beinhalten diese keinen Typen eines SharedPreferences-Objektes, erübrigt sich eine weitere Behandlung. Andernfalls wird die Hauptuntersuchung auf den Einheiten (Units), also den Jimple-Ausdrücken der Methode, ausgeführt. Darin befindliche Aufrufe werden hinsichtlich ihres Namens, ihrer deklarierenden Klasse und dem Typ ihres Rückgabewertes gefiltert und gefundene Zugriffe auf SharedPreferences-Objekte als Objekte der Klassen File, Entry und Access vorgehalten, um abschließend in das übergebene Modell übertragen zu werden. Weiterhin ist eine Übersicht der Ergebnisse als Textdatei ausgebbar.

3.2.2 Analyse-Klassen

Die im Java-Paket `org.schoenborn.thesis.soot.plugins.android.sharedpreferences` beinhalteten Klassen `SharedPreferencesAnalysis`, `SharedPreferencesTransformer` und `SharedPreferencesPlugin` ermöglichen die Durchführung der Analyse. Die Klasse `SharedPreferencesAnalysis` bildet den Kern der Untersuchung. Ihre Einbindung als Phase in den Gesamtablauf des Soot-Aufrufs erfolgt über die Hilfsklassen `SharedPreferencesPlugin` und `SharedPreferencesTransformer`. Erstere definiert als Implementierung eines `AbstractPhasePlugin` neben der Beschreibung des Plugins zusätzliche Optionen zur Übergabe externer Parameter und zur Verwendung der Zweiteren. Als `SceneTransformer` umgesetzt, bekommt diese die Parameter `mapper-phase`, `model-phase` und `results-output-logfile` injiziert um mit diesen, ergänzt durch die `scene`, die Analyse vom Typ `SharedPreferencesAnalysis` zu initiieren.

SharedPreferencesAnalysis

Neben einer `org.slf4j.Logger`-Instanz zur Formatierung und Ausgabe von Statusmeldungen enthält die Klasse als Attribute ein Modell, einen Mapper, eine Soot-Szene und die Analyseergebnisse. Das Modell (`private AndroidDevice model`) dient der Aggregation von Ergebnissen der verschiedenen Phasen und soll auch durch diese Analyse ergänzt werden. Der Mapper (`private ComponentCallgraphMapper mapper`) ordnet dafür Komponenten anhand in ihnen enthaltener Methoden zu. Die Soot-Szene (`private Scene scene`) enthält sämtliche Soot zum Analysezeitpunkt bekannten Informationen und Zwischenrepräsentationen der App. Die Analyseergebnisse werden in Form eines Satzes von gefundenen Dateien (`private Set<File> results`) vorgehalten. Mit Ausnahme der leer initiierten Ergebnisse werden der Analyseklassse ihre Attribute bei der Initiierung über den Aufruf aus dem `SharedPreferencesTransformer` übergeben. Wie schon zu Beginn des Abschnittes erwähnt, verarbeitet sie die Jimple-Repräsentation der zu untersuchenden App. Dazu läuft sie über sämtliche Methoden aller Soot bekannten, aktiven Klassen und prüft diese in einer Voruntersuchung zunächst aufeine mögliche Verwendung von SharedPreferences-Objekten. Im Falle eines positiven Ergebnis wird die Methode einer Hauptuntersuchung unterzogen.

Voruntersucherung Das Selektionskriterium für eine weitere Untersuchung der betrachteten Methode ist das Vorkommnis von bestimmten Typen in den Deklarationen ihrer lokalen Variablen (Locals). Konkret handelt es sich dabei sowohl um die SharedPreferences, in der Notation `android.content.SharedPreferences`, und den SharedPreferences Editor, in der Notation `android.content.SharedPreferences$Editor`. Die Überprüfung dieses Kriteriums erfolgt über die Methode `private boolean localsContainType(Collection<Local> locals, String type)`. Gleicht der Typ einer lokalen Variable diesem übergebenen String, liefert jene `true` zurück.

Hauptuntersucherung Nach positivem Ergebnis der Voruntersuchung erfolgt die Hauptuntersuchung mit dem Aufruf der Methode `private void extractPreferences(SootMethod method, boolean callsPrefs, boolean editsPrefs)`. Dort werden in der übergebenen Methode enthaltene Jimple-Ausdrücke (`Stmt`), welche in Soot-Einheiten (`Unit`) gekapselt sind, anhand einer Reihe von Kriterien gefiltert. In einer Methode gefundene Dateien, Einträge und Zugriffe werden in der `Map<String, File> filesbyRef` eingetragen und verwaltet, als Schlüssel dienen die lokalen Jimple-Variablennamen. Ergänzend dazu nimmt `Map<String, String> editorMapping` Zuordnungen zwischen SharedPreferences und mög-

lichen SharedPreferences-Editor Instanzierungen auf. Als Schlüssel werden die Bezeichnungen der lokalen Jimple-Variablen von Editor-Aufrufen eingetragen, die korrespondierenden Werte sind je die Jimple-Variablennamen der verwendeten Datei. Grundsätzlich werden nur Ausdrücke, welche einen Aufruf beinhalten, untersucht. Diese sind über ihren Typ (`InvokeExpr`) oder den Aufruf von `boolean soot.jimple.Stmt.containsInvokeExpr()` erkennbar. Für sie werden als Eigenschaften bestimmt:

1. Name der aufgerufenen Methode
   `String name = invokeExpr.getMethod().getName();`
2. Name der deklarierenden Klasse
   `String declaringClass =`
   `invokeExpr.getMethodRef().declaringClass().getName();`
3. Typ des Rückgabewertes
   `String returnType =`
   `invokeExpr.getMethodRef().returnType().toString();`

Je nach Ausprägung werden vier Fälle exklusiv unterschieden und behandelt:

1. Instanzierung einer SharedPreferences-Datei
   Bedingung: Der Rückgabetyp gleicht `android.content.SharedPreferences`.
   Aktion: Nach der Bestimmung von `prefKey` wird versucht, den Paketnamen aus dem Aufruf zu lesen und mit diesem ein `File`-Objekt instanziert. Die dazu verwendete Methode `fromInvoke` wird in Abschnitt 3.2.3, mit der sie bereitstellenden Klasse, erläutert. Die Repräsentation der Datei wird zuletzt mit ihrem Schlüssel `prefKey` in `filesbyRef` eigetragen.
2. Instanzierung eines Editors auf einer Datei
   Bedingung: Die aufgerufene Methode heißt `edit`, die deklarierende Klasse gleicht `android.content.SharedPreferences` und der Rückgabetyp gleicht `android.conte⌋nt.SharedPreferences$Editor`.
   Aktion: Nach Bestimmung des Variablennames der zu editierenden Datei `prefKey` und dem des Editors `editorRef` werden beide in `editorMapping` eigetragen, um eine Assoziation von Schreibzugriffen auf diese Datei zu gewährleisten.
3. Lesezugriff auf einen Eintrag
   Bedingung: Die aufgerufene Methode heißt nicht `edit` und die deklarierende Klasse gleicht `android.content.SharedPreferences`.
   Aktion: Der vorausgehenden Bestimmung von `prefKey` folgt ein Versuch der Instan-

zierung eines `Entry`-Objektes über dessen Methode `fromInvoke`, welche in Abschnitt 3.2.3 beschrieben werden. Die Repräsentation des Eintrages wird danach in der zugehörigen, unter `prefKey` in `filesbyRef` eigetragenen Datei vermerkt.

4. Schreibzugriff auf einen Eintrag
   Bedingung: Die aufgerufene Methode heißt weder `clear` noch `commit` und die deklarierende Klasse gleicht `android.content.SharedPreferences$Editor`.
   Aktion: Die gleiche Vorgehensweise wie bei der Behandlug des Lesezugriffs.

Sind alle Ausdrücke einer Methode abgearbeitet, werden die lokalen, in `filesbyRef` vorgehaltenen Ergebnisse in die Analyseergebnisse überführt. Noch nicht in den `results` enthaltene Dateien werden diesen hinzugefügt. Bereits dort auffindbare Dateien werden um zuvor nicht bekannte Einträge oder Zugriffe aus den lokalen Befunden ergänzt. Die dafür notwendigen Vergleichs- und Zusammenführungsmethoden werden in Abschnitt 3.2.3, unter dem Punkt „Gemeinsamkeiten“, ausgeführt.

Ausgabe in Logdateien Ist der bei Initialisierung der Analyse übergebene Parameter `resultsLogfile` ungleich `null`, wird ein Logger zur Ausgabe der Ergebnisse erzeugt. Die dafür zuständige Methode (`private Logger createResultsLogger(String file)`) setzt dabei Optionen für die Formatierung der Logereignisse und für die Zieldatei unter dem Pfad `file` und gibt das instanzierte `Logger`-Objekt zurück. Abschließend wird unter Übergabe des `Logger` auf jeder in den Ergebnissen enthaltenen Datei die Methode `logResult()` (vergleiche ebenso Abschnitt 3.2.3, „Gemeinsamkeiten“) aufgerufen.

Annotation des Modells Die finale Übertragung der Untersuchungsergebnisse in das Modell übernimmt die Methode `private void commitResultsToModel()`. Die in `results` abgelegten `File-`, `Entry-` und `Access-`Objekte werden von ihr in einer dreifach geschachtelten Schleife durchlaufen, wobei für jedes die `exportModelData()`-Methode aufgerufen wird (siehe ebenfalls 3.2.3, „Gemeinsamkeiten“). Nachdem deren Ausgabe jeweils mit der Ausgabeversion des übergeordneten Objektes verknüpft worden ist, wird die Datenstruktur an das Modell (`model`) angehängt: Gemäß dem erweiterten Modell aus Abschnitt 3.1 wird eine Datei mit dem Gerät oder der jeweiligen App verknüpft. Zugriffe werden mit einer Komponente verknüpft, wenn sie sich der Methode, welche den Zugriff beinhaltet, über den `ComponentCallgraphMapper mapper` zuordnen lassen.

Veranschaulichung von Jimple-Ausdrücken

```
1  private void loadFromPreferences() {
2    SharedPreferences preferences = getSharedPreferences("editPrefs",
   ↪  MODE_PRIVATE);
3    String string = preferences.getString("string", "Default String value.");
4  }
```

```
1  {
2    org.schoenborn.thesis.android.benchmarkapp.MainActivity $r0;
3    android.content.SharedPreferences $r1;
4
5    $r0 := @this: org.schoenborn.thesis.android.benchmarkapp.MainActivity;
6
7    $r1 = virtualinvoke
   ↪  $r0.<org.schoenborn.thesis.android.benchmarkapp.MainActivity:
   ↪  android.content.SharedPreferences
   ↪  getSharedPreferences(java.lang.String,int)>("editPrefs", 0);
8
9    interfaceinvoke $r1.<android.content.SharedPreferences: java.lang.String
   ↪  getString(java.lang.String,java.lang.String)>("string", "Default String
   ↪  value.");
10
11   return;
12 }
```

Listing 3.1: Beispiel zur Jimple-Zwischenrepräsentation

Zur Illustration der beschriebenen Vorgehensweise und den Ausführungen zu den Analyseklassen (weiter unten) soll Listing 3.1 dienen. Es enthält als ersten Teil den Quelltext der Methode `loadFromPreferences()` aus der Komponente `MainActivity` der App `org.schoenborn.thesis.android.benchmarkapp`. Die von Soot aus dem Dalvik-Bytecode generierte Jimple-IR dazu wird direkt darunter dargestellt. Die lokalen Variablen (Locals) werden in Zeile 2 und 3 deklariert und die zweite davon würde im Zuge der Analyse eine weitere Untersuchung der Methode signalisieren. Bei den Units beziehungsweise, da eine Jimple-Repräsentation betrachtet wird, Ausdrücken korrespondieren die Zuweisung auf $r1 (Zeile 7) mit der Java-Zuweisung von `preferences` und der Aufruf auf $r1 (Zeile 9) mit der Java-Zuweisung von `string`. Eine zu untersuchende Anweisung wie in Zeile

9 besteht aus den Bestandteilen Befehl (`interfaceinvoke`), Definition des Objektes auf dem der Befehl erfolgt ($r1, siehe Zeile 7), deklarierende Klasse, Rückgabetyp, Name, Typen der Parameter und den Werten der Parameter. Festzustellen ist, dass sich die Variable `string` selbst nicht mehr in der Jimple-Form wiederfindet. Begründen lässt sich dies dadurch, dass sie mangels weiterer Verwendung während der Optimierung durch den Compiler weggelassen wurde.

3.2.3 Klassen für Dateien, Einträge und Zugriffe

Die im Java-Paket `org.schoenborn.thesis.soot.plugins.android.sharedpreferences.model` beinhalteten Klassen `File`, `Entry` und `Access` dienen der Speicherung und Verarbeitung von Zwischenergebnissen der Analyse und bilden je eine gefundene SharedPreferences-Datei, einen Eintrag in einer solchen und einen Zugriff auf einen Eintrag ab. Hinsichtlich ihrer Attribute und Beziehungen zueinander gleichen sie den ihnen zugrundeliegenden Klassen zur Erweiterung des Modells aus Abschnitt 3.1. Diese strukturelle Nähe hat den Vorteil, dass sie die abschließende Übertragung der Ergebnisse auf das Modell, in Form der Annotation desselben, vereinfacht. Ebenfalls enthalten sind die `enum`-Klassen `FileMode`, `EntryType` und `AccessMode` zur Abbildung je einer Eigenschaft der obigen. Zum Vorhalten der gewonnenen Informationen weist jede der drei Klassen diese und weitere entsprechende Attribute auf.

File Wie die zuvor vorgestellte gleichnamige Klasse zur Erweiterung des Modells repräsentiert `File` eine SharedPreferences-Datei mit den Attributen Name (`private String name`), Modus (`private FileMode mode`) und Pfad (`private String path`) sowie einem Satz von Einträgen (`private Set<Entry> entries`), welcher die Aggregationsbeziehungen zu diesen realisiert. Der für den Modus genutzte Typ `FileMode` wird wie der analog dazu im Ecore-Modell verwendete `OperatingMode` eingesetzt und als Java `enum`-Objekt umgesetzt. Neben den gültigen Werten enthält er eine Methode `public static FileMode byInt(int i)` zur Zuordnung jener zu den in der Arndoid-API definierten und in Abschnitt 2.1.3 aufgeführten Konstanten für die Zugriffsberechtigungen. Die durch die Klasse bereitgestellte statische Methode `public static File fromInvoke(InvokeExpr invokeExpr, String packageName)` liefert die im übergebenen Jimple-Ausdruck initialisierte SharedPreferences-Datei als neues `File`-Objekt. Unter Berücksichtigung der in den Grundlagen (siehe Abschnitt 2.1.3) beschriebenen drei unterschiedlichen, gültigen Aufrufsmöglichkeiten wird die `invokeExpr` zerlegt, um die Attribute Name, Modus und Pfad zu erhalten.

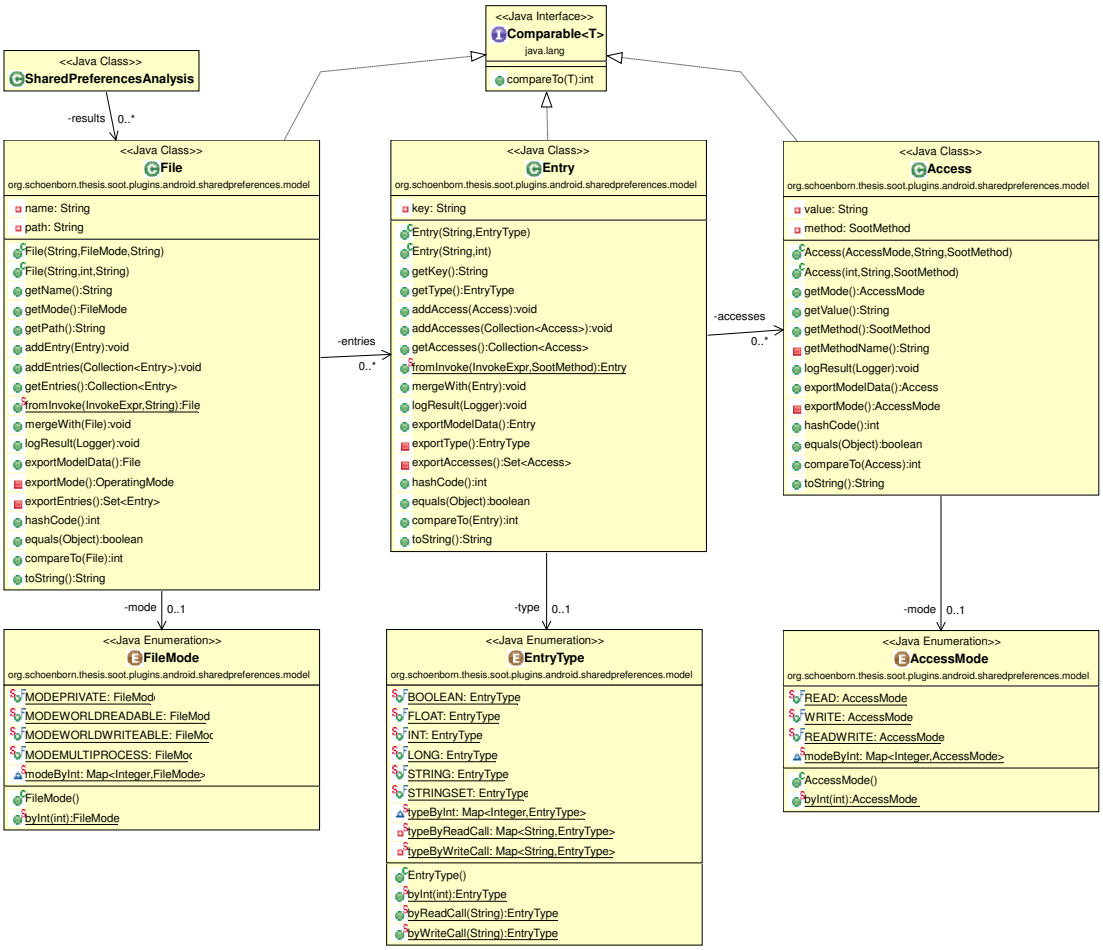


Abbildung 3.3: UML-Darstellung der Implementierung (Datentyp-Klassen)

Entry Einträge in einer SharedPreferences-Datei werden durch die Klasse `Entry` modelliert. Ihre Attribute sind der Eintragsschlüssel (`private String key`) und -typ (`private EntryType type`) sowie ein Satz von Zugriffen (`private Set<Access> accesses`). Der Typ des Attributes `type` wird wie der im Modell gleichnamige `EntryType` verwendet und ebenfalls mittels Java-`enum`-Objekt realisiert. Außer seinen gültigen Werten beinhaltet jener zwei Methoden zur Zuordnung dieser Werte zu den Namen von Lese- (`public static EntryType byReadCall(String s)`) oder Schreibaufrufen (`public static EntryType byWriteCall(String s)`) auf die SharedPreferences. Mögliche Ausprägungen sind in Abschnitt 2.1.3 beschrieben. Zur Erzeugung aus Aufrufsausdrücken dient die statische Methode `public static Entry fromInvoke(InvokeExpr invokeExpr, SootMethod method)`. Hierbei findet eine Fallunterscheidung nach Lese- und Schreibzugriffen statt, welche über den Methodennamen (siehe Abschnitt 2.1.3) per `EntryType.byReadCall()` und `En`

`tryType.byWriteCall()` getroffen wird. Aus dem Aufruf lassen sich Schlüssel und Wert extrahieren, womit mit ersterem der Eintrag und mit zweiterem ein zugehöriger Zugriff erzeugt wird.

Access Zugriffe auf Einträge in SharedPreferences-Dateien werden durch die Klasse `Access` dargestellt. Neben den Attributen für den Zugriffsmodus (`private AccessMode mode`) und -wert (`private String value`) beinhalten Objekte der Klasse zusätzliche eine Referenz (`private SootMethod method`) auf die Methode, welche den Aufruf durchführt. Der Typ `AccessMode` von `mode` findet sich so ebenfalls im Modell wieder.

Gemeinsamkeiten

Bei Betrachtung der Klassen `File`, `Entry` und `Access` fällt auf, dass es Gemeinsamkeiten in der Funktionalität gibt und sich einige einander entsprechende Methoden finden lassen.

Zugänglichkeit von Eigenschaften Es handelt sich bei den Attributen aller drei Klassen konventionellerweise um private Felder, wodurch ein Zugriff über Aufrufe aus Objekten anderer Klassen nicht möglich ist. Um notwendige Überprüfungen, etwa in Vergleichen, dennoch vornehmen zu können, existiert zu jedem Attribut `private T value` eine Ausgabemethode `public T getValue()`.

Vergleichbarkeit und Sammlungen Zur vereinfachten Handhabung der Typen und zur Verwendung in Sammlungen (`java.util.Collection`), wie `Map` und `Set`, implementieren die Klassen die `java.lang.Comparable<T>` Schnittstelle non-generisch. Damit sind Vergleiche von Objekten identischen Types untereinander möglich. Die durch die Schnittstelle geforderten Methoden zum Vergleich (`public int compareTo(T other)`), zur Feststellung der Gleichheit (`public boolean equals(Object o)`) und zur Berechnung eines Hashcodes (`public int hashCode()`) werden dementsprechend zusätzlich überschrieben. Zu beachten ist hier, dass die Vergleichsmethode bei `File`- und `Entry`-Objekten ohne Berücksichtigung der beinhalteten Sets von `Entry`- respektive `Access`-Objekten erfolgt.

Vereinigung von Objekten Da die Untersuchung intra-prozedural erfolgt, können Zugriffe auf die gleiche Datei oder den gleichen Eintrag einer Datei in verschiedenen Methoden zur Bildung von Duplikaten in dem Zwischenspeicher für die Ergebnisse führen. Vor

Abschluss der Analyse müssen diese vereint werden, so dass alle Zugriffe auf gleiche Einträge und alle Zugriffe auf gleiche Dateien je einem Objekt zugeordnet sind. Dazu addiert nach festgestellter Gleichheit (siehe oben) je eine Methode `public void mergeWith(T t)` beide `entries-` beziehungsweise `accesses`-Sets miteinander.

Textausgabe für Meldungen und Ergebnisse Neben der, zwecks übersichtlicherer Ausgabe während der Analysedurchführung, überschriebenen `public String toString()` Methode, existiert jeweils eine Methode `public void logResult(Logger RESULT)` für die textuelle Darstellung der Ergebnisse. Letztere hängt jeweils die relevanten Eigenschaften des Objektes in Form eines Loggingereignisses in abweichender Formatierung an den übergebenen Logger an und ruft sich rekursiv für je alle in `entries` (`File`) und `accesses` (`Entry`) enthaltenen Objekte auf. Listing 3.2 zeigt wie sich die Ausgabe gliedert. Unter den erkannten SharePreferences-Dateien (Zeilen 2 und 6) werden Einträge mit Datentyp (Zeilen 3 und 7) angeführt. Die Zugriffe auf die Einträge sind jeweils zeilweise und eingerückt unter diesen angeordnet.

```
1 APP: com.example.android.app
2 somePreferences.xml (MODEPRIVATE)
3 > userName - STRING
4     getUserName.READ("")
5     setUserName.WRITE("default")
6 otherPrefernces.xml (MODEPRIVATE)
7 > isSetup - BOOLEAN
8     initializeApp.READ("")
```

Listing 3.2: Beispiel zur Formatierung von Ergebnislogs

Export in das Ecore-Modell Für die Annotation des Ecore-Modells ist eine Umwandlung der Objekte in die Klassen, welche aus der vorherigen Erweiterung des Modells resultieren, notwendig. In jede der Klassen wurde dazu eine Methode `exportModelData()` integriert. In der vollen Signatur (mit dem vollständigen Paketnamen des Zieldatentypes) schreibt sich diese beispielhaft für `File`: `public de.uni_bremen.st.model.android.devi⌋ce.sharedpreferences.File exportModelData()`. Nach Instanzierung des Ausgabeobjektes durch eine Factoryinstanz werden jeweils die Attribute gesetzt, wobei das jeweilige `Enum`-Attribut noch über eine private Hilfsmethode zugeordnet wird. Dies ist, wieder exemplarisch für `File`, dann `private de.uni_bremen.st.model.android.device.shared⌋preferences.OperatingMode exportMode()`.

4 Evaluation

An dieser Stelle wird die im vorherigen Kapitel vorgestellte Analysekomponente durch eine Kombination von zwei Verfahren evaluiert. Das erste davon, ein eigens hierfür entwickelter Micro-Benchmark, soll alle relevanten, unterscheidbaren Fälle der lokalen Verwendung von SharedPreferences in Methoden von Android-Apps abdecken. Anschließend wird eine stichprobenartige Fallstudie mit einigen Apps durchgeführt. Bei diesen handelt es sich um reale Apps aus dem Google Play Store, die unabhängig vom Autor dieser Arbeit entstanden sind. Auf die Überlegungen zur Auswahl wird in Abschnitt 4.2 eingangen.

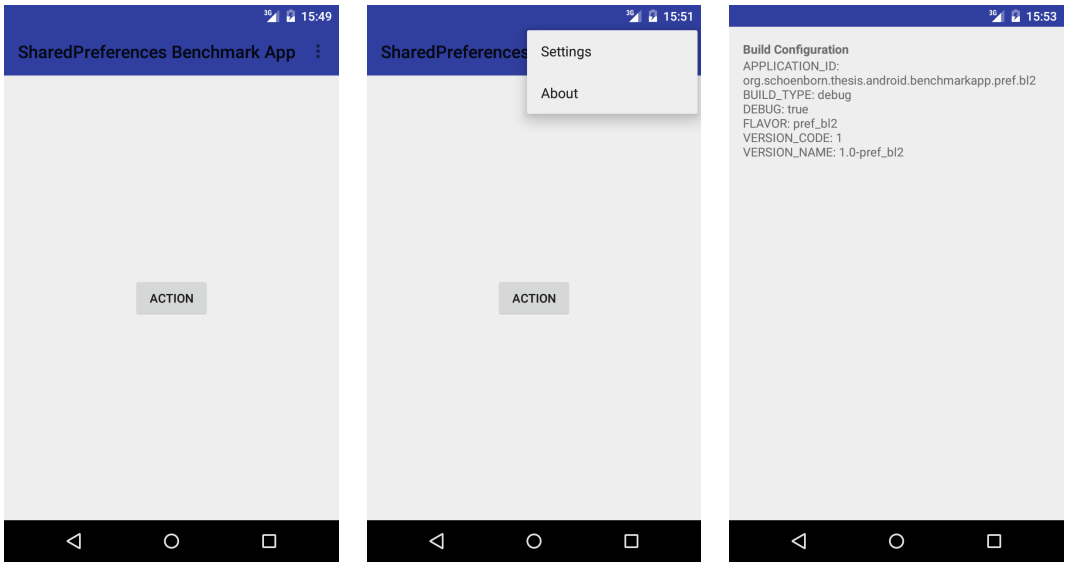
4.1 Benchmark

Als erster Bestandteil der Evaluation wird ein Systemtest durchgeführt, welcher anhand eines Satzes von Fällen eine Bewertung der Genauigkeit der untersuchten Analyse ermöglicht. Als Testdaten dient ein Satz von eigens entwickelten Apps in Binärform. Aus der angestrebten Vollständigkeit in Bezug auf sowohl als Zugriffe zu erkennenden Fälle als auch, in Abgrenzung dazu, nicht erkennbaren Fälle begründet sich die Bezeichnung als Benchmark (im Sinne von Maßstab).

4.1.1 Umsetzung

Die einzelnen Fälle für den Benchmark wurden als Android-Apps implementiert. Im Hinblick auf die Verwendung als Benchmark-Fall sind diese möglichst minimal aufgebaut. Bei Ausführung bietet sich dem Nutzer die Möglichkeit vom Hauptbildschirm (Abbildung 4.1a) aus, über eine mit *Action* gekennzeichnete Schaltfläche, eine Operation auf SharedPreferences-Dateien aufzurufen. Dazu erfolgt eine visuelle Bestätigung in Form eines Text-Popups am unteren Bildschirmrand. Über das Menü-Symbol oben rechts werden die Auswahlmöglichkeiten *Settings* und *About* angeboten (Abbildung 4.1b), erstere ist bei den Testfällen nicht funktional, während zweitere zu einer Ausgabe der

Build-Configuration der App (Abbildung 4.1c) führt. Während die Apps theoretisch auf Android-Versionen ab 2.3 Gingerbread (API Level 10, bezogen auf das Android Framework [15]) lauffähig sind, wurden sie im Zuge der Entwicklung tatsächlich nur auf 5.0 und 5.1 Lollipop (API Level 21 und 22 [15]) ausgeführt.



(a) MainActivity (b) MainActivity (Menü) (c) AboutActivity

Abbildung 4.1: Screenshots der Benchmark-App (Fall pref_bl2)

Entwicklungsumgebung Die verschiedenen Testfälle unterscheiden sich lediglich durch ihre Bezeichnung und die konkrete Implementierung von wenigen Methoden. Damit besteht eine große, gemeinsame Menge von Quelltexten und Ressourcen, welche eine manuelle Pflege beziehungsweise getrennte Entwicklung aller Benchmark-Apps umständlich und, hinsichtlich späterer Änderungen auf vielen identischen Quellen, möglicherweise fehleranfällig gemacht hätten. Praktischerweise bietet die verwendete, im offiziellen Android Software Development Kit (SDK) [6] enthaltene Android Studio Integrated Development Environment (IDE) [3] Mechanismen, um mit solchen Szenarien umzugehen. Dazu sieht das voreingerichtete Gradle Buildsystem die Konfiguration von als Flavors bezeichneten Produktvarianten vor, beispielsweise zur Erzeugung von Demo- und Vollversionen von Apps. Zur Erzeugung verschiedener Flavors einer App ist es nötig, diese in der Build-Konfigurationsdatei anzugeben und für jeden Flavor zusätzliche Verzeichnispfade für dessen spezifische Quelltexte anzulegen. Listing 4.1 zeigt einen Auszug aus der Konfiguration für die Benchmark-Apps für den Flavor *nopref* (Zeile 13-17) und den Flavor *pref_bl2* (Zeile 157-161). Dabei werden je die Application-Identifier (ID), welche als ein-

zigartiges Identifizierungsmerkmal veröffentlichter Apps dient, sowie der Anwendungsname als String-Ressource und der Versionsname überschrieben. Detaillierte Informationen zur Konfiguration von Gradle und eine exemplarische Erläuterung der Vorgehensweise zur Implementierung von Flavors finden sich in der offiziellen Dokumentation [5].

```
21  productFlavors {
22      // Initial app skeleton
23      nopref {
24          applicationId "org.schoenborn.thesis.android.benchmarkapp.nopref"
25          resValue "string", "app_name", "SPB nopref"
26          versionName "1.0-nopref"
27      }

156     // 2 accesses, 1 entries, 1 file
157     pref_bl2 {
158         applicationId "org.schoenborn.thesis.android.benchmarkapp.pref.bl2"
159         resValue "string", "app_name", "SPB Local Case 2"
160         versionName "1.0-pref_bl2"
161     }
```

Listing 4.1: Konfiguration von Flavors in Gradle (Auszüge)

App-Architektur Bezogen auf die Nutzung als Benchmark-App mit jeweils unterschiedlichem Verhalten hinsichtlich der Verwendung von SharedPreferences-Einträgen ist minimal eine Android-Aktivität zu implementieren gewesen. Neben dieser `MainActivity`, welche auch Startaktivität jeder App ist, sind noch die Klassen `MainActivityOneButton` und `AboutActivity` enthalten. Abbildung 4.2 visualisiert deren Beziehungen. Die beiden letztgenannten Klassen sind nicht Flavor-spezifisch, so dass für jeden Testfall nur je eine von `MainActivityOneButton` erbende `MainActivity` programmiert werden musste. Aus der überschriebenen Methode `onClickButtonAction` waren darin jeweils noch ein oder mehrere Aufrufe auf Methoden zur eigentlichen Manipulation von SharedPreferences einzusetzen. Die einzige Ausnahme bildet der Fall *pref_ bl7* (siehe Tabelle 4.2), welcher zusätzlich einen Dienst zum Zugriff auf SharedPreferences verwendet.

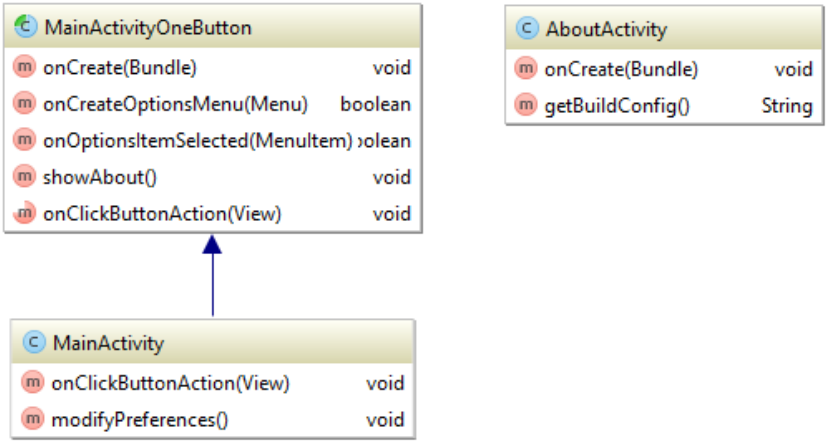


Abbildung 4.2: UML-Darstellung der Benchmark-App Activities (pref_bl2)

4.1.2 Fälle

Die für den Benchmark implementierten Fälle lassen sich, trotz ihrer architekturellen Nähe zueinander, in zwei Gruppen einteilen. Die erste Gruppe bildet eine Reihe von, im Bezug auf die Verwendung von SharedPreferences, einfachen Fällen. Die zweite Gruppe bietet dazu ergänzend weitere, komplexere Szenarien, die der tatsächlichen Verwendung von SharedPreferences in einer realen App näherkommen.

Einfache Fälle Als Grundüberlegung für die zu erkennenden Fälle wurde, in Anlehnung an das Black-Box Prinzip ohne Berücksichtigung der Implementierung, zunächst eine Übersicht aller möglichen Einzelzugriffe bzw. atomaren Testfälle aufgestellt. Auf Basis der Aufrufs- und Parameterkombinationen ergeben sich folgende Möglichkeiten der Variation eines Zugriffs auf einen SharedPreferences-Eintrag:

1. Aufruf:
   getDefaultSharedPreferences (Name implizit, Modus implizit),
   getPreferences (Name implizit, Modus explizit),
   getSharedPreferences (Name explizit, Modus explizit)
2. Modus: Multi Process, Private, World Readable, World Writeable
3. Zugriff: Lesen, Schreiben
4. Eintragstyp: Boolean, Float, Int, Long, String, Set<String>

Durch Bildung aller möglichen Kombinationen der Aufrufe mit den Merkmalen Modus, so vorhanden, Art des Zugriffes und sämtlicher Datentypen für die Einträge ergäben sich 108 prüfbare Zusammenstellungen. Da aber eine solche Reihe von Benchmarkfällen aufgrund der Ähnlichkeit untereinander und der Praxisferne nicht zu einem aussagekräftigen Ergebnis führt, wurden diese unter Kombination einzelner Merkmale zusammengefasst. Eine vollständige Übersicht dieser Benchmarkfälle findet sich in Tabelle 4.1. Die zwölf Fälle *pref_drb* bis *pref_dwss* beinhalten je einen Lese- oder Schreibzugriff auf einem der unterstützten Eintragstypen. Die vier Fälle *pref_mmm* bis *pref_mww* beinhalten je einen Lese- und Schreibzugriff mit einer der vier Dateiberechtigungen. Der letzte Fall (*pref_mnxm*) ergänzt die noch nicht behandelte fehlende Variation der Dateinamen.

Komplexe Fälle Bei diesen Fällen kommen unterschiedliche Kontroll- und Datenflussszenarien zum Tragen. Dabei sollten die Fälle *pref_bl1* bis *pref_bl7* erkannt werden, da sämtliche Informationen zu den SharedPreferences-Aufrufen lokal innerhalb der Methode vorliegen. Dagegen stehen die als Abgrenzung dienenden Fälle *pref_bnl1* bis *pref_bnl3*, bei welchen Informationen außerhalb der Methode definiert werden. In Tabelle 4.2 sind alle komplexen Benchmarkfälle aufgelistet und kurz beschrieben.

Fall	D	E	Z	Kurzbeschreibung
pref_drb	1	1	1	Die App liest einen Eintrag vom Typ `boolean`.
pref_drf	1	1	1	Die App liest einen Eintrag vom Typ `float`.
pref_dri	1	1	1	Die App liest einen Eintrag vom Typ `int`.
pref_drl	1	1	1	Die App liest einen Eintrag vom Typ `long`.
pref_drs	1	1	1	Die App liest einen Eintrag vom Typ `String`.
pref_drss	1	1	1	Die App liest einen Eintrag vom Typ `Set<String>`.
pref_dwb	1	1	1	Die App schreibt einen Eintrag vom Typ `boolean`.
pref_dwf	1	1	1	Die App schreibt einen Eintrag vom Typ `float`.
pref_dwi	1	1	1	Die App schreibt einen Eintrag vom Typ `int`.
pref_dwl	1	1	1	Die App schreibt einen Eintrag vom Typ `long`.
pref_dws	1	1	1	Die App schreibt einen Eintrag vom Typ `String`.
pref_dwss	1	1	1	Die App schreibt einen Eintrag vom Typ `Set<String>`.
pref_mmm	1	2	2	Die App liest und schreibt je einen Eintrag auf den SharedPreferences mit `MODE_MULTI_PROCESS`-Berechtigung.
pref_mp	1	2	2	Die App liest und schreibt je einen Eintrag auf den SharedPreferences mit `MODE_PRIVATE`-Berechtigung.
pref_mwr	1	2	2	Die App liest und schreibt je einen Eintrag auf den SharedPreferences mit `MODE_WORLD_READABLE`-Berechtigung.
pref_mww	1	2	2	Die App liest und schreibt je einen Eintrag auf den SharedPreferences mit `MODE_WORLD_WRITEABLE`-Berechtigung.
pref_mnxm	4	4	4	Die App liest und schreibt auf den SharedPreferences unter vier unterschiedlichen Namen und Berechtigungen.

Legende: Die Buchstaben D, E und Z sind als Abkürzungen für SharedPreferences-Datei, -Eintrag und -Zugriff zu verstehen. Die Werte dieser drei Spalten beziehen sich auf ihre jeweilig pro App verwendete Anzahl. In der Spalte „Kurzbeschreibung“ wird das Verhalten der App bei Aufruf von *onClickButtonAction* zusammengefasst.

Tabelle 4.1: Übersicht einfacher Benchmarkfälle

Fall	D	E	Z	Kurzbeschreibung
pref_bl1	1	2	2	Die App schreibt lokal in die SharedPreferences, dabei sind alle Parameter im Aufruf definiert.
pref_bl2	1	1	2	Die App liest und schreibt lokal in die SharedPreferences, wobei eine kompaktere, verkettete Notation der Aufrufe verwendet wird.
pref_bl3	1	1	2	Die App liest und schreibt lokal in die SharedPreferences. Alle Parameter wurden dabei zuvor als Variablen deklariert.
pref_bl4	2	2	2	Die App liest und schreibt lokal in die SharedPreferences und die Parameter sind dabei je teilweise im Aufruf enthalten und zuvor als Variablen deklariert worden.
pref_bl5	1	1	3	Die App liest und schreibt lokal in die SharedPreferences. Die Parameter wurden dabei zuvor als Variablen deklariert und zwischen den Aufrufen neu zugewiesen.
pref_bl6	1	3	3	Die App liest lokal aus den SharedPreferences, wobei die Parameter aus komplexen Datenstrukuren (Array und Liste) gelesen werden.
pref_bl7	1	1	2	Die App startet für Lese- und Schreibzugriffe auf SharedPreferences einen Dienst, in welchem die Verwendung lokal erfolgt.
pref_bnl1	2	2	2	In der App werden für die durchzuführenden Lesezugriffe auf SharedPreferences nötige Parameter von außen an die Methode übergeben.
pref_bnl2	2	2	2	In der App werden für die durchzuführenden Lesezugriffe auf SharedPreferences nötige Parameter über Methodenaufrufe nicht-lokal erzeugt.
pref_bnl3	1	1	2	In der App werden, für die durchzuführenden Lese- und Schreibzugriffe auf SharedPreferences, nötige Parameter aus Klassenattributen nicht-lokal gelesen.

Legende: Die Buchstaben D, E und Z sind als Abkürzungen für SharedPreferences-Datei, -Eintrag und -Zugriff zu verstehen. Die Werte dieser drei Spalten beziehen sich auf ihre jeweilig pro App verwendete Anzahl. In der Spalte „Kurzbeschreibung“ wird das Verhalten der App bei Aufruf von *onClickButtonAction* zusammengefasst.

Tabelle 4.2: Übersicht komplexer Benchmarkfälle

4.1.3 Ergebnisse

Bei der Durchführung des Benchmarktests ergeben sich die in Tabelle 4.3 dargestellten Ergebnisse. Über die Gesamtmenge der Benchmarkfälle betrachtet wurden 37 Zugriffe mit lokalem Bezug der Werte ihrer Parameter auf SharedPreferences-Einträge korrekt erkannt (True Positive). Neben der Nichterkennung von drei weiteren solcher Einträge für den Fall *pref_ bl6* (False Negative) kam es für den Fall *pref_ bnl3* auch zur fälschlichen Erkennung von zwei Zugriffen mit nicht-lokalem Bezug der Werte ihrer Parameter (False Positives). Damit wurden ein Precision-Wert von rund 0,95 und ein Recall-Wert von rund 0,93 erreicht. Bei Nichtberücksichtigung der einfachen Fälle ergeben sich für die verbleibenden zehn Fälle *pref_ bl1* bis *pref_ bnl3* 13 True Positives, vier True Negatives, zwei False Positives und drei False Negatives. Es leite sich daraus ein Precision-Wert von rund 0,86 und ein Recall-Wert von rund 0,82 ab. Von besonderem Interesse sind nach der Durchführung des Benchmarktest die Fälle *pref_ bl6* und *pref_ bnl3*, da die automatische Analyse zu diesen unerwartete Resultate lieferte. Bei *pref_ bl6* ist dies durch die Tatsache zu erklären dass sowohl Schlüssel als auch Werte für die SharedPreferences-Aufrufe in komplexeren Datenstrukturen enthalten sind. Konkret wurden die Schlüssel in einem Array und die Werte in einer verknüpften Liste hinterlegt. Da die Zuordnung der Instruktionen zur Entnahme der Werte und ihrer zugehörigen Einspeicherungsbefehle in dem Analyseverfahren nicht vorgesehen ist, schlägt die Erkennung der Zugriffe fehl. Bei *pref_ bnl3* werden hingegen Zugriffe erkannt, obwohl die Parameter eines SharedPreferences-Aufrufe außerhalb der lokalen Methode, als Attribute der Klasse, definiert wurden. Nach Begutachtung der decompilierten Quellen lässt sich dieses Verhalten eindeutig durch Compiler-Optimierungen erklären, denn auch dort stehen die Werte der Parameter lokal in dem Aufruf. Die Fälle *pref_ drss*, *pref_ dwss* und *pref_ bl4*, bei denen einzelne Parameter nicht erkannt wurden (statt des Wertes wird eine Soot Variable angezeigt), lassen sich auf die Verwendung von komplexeren Datenstrukturen oder Neuzuweisungen von Variable zurückführen. Eine Relativierung und Diskussion der Ergebnisse, auch in Verbindung mit den Ergebnissen der Fallstudie, ist Bestandteil des, auf diese folgenden Kapitels 5.

Fall	D	Güte	E	Z	Güte
pref_drb	1/1	2/2	1/1	1/1	2/2
pref_drf	1/1	2/2	1/1	1/1	2/2
pref_dri	1/1	2/2	1/1	1/1	2/2
pref_drl	1/1	2/2	1/1	1/1	2/2
pref_drs	1/1	2/2	1/1	1/1	2/2
pref_drss	1/1	2/2	1/1	1/1	1/2
pref_dwb	1/1	2/2	1/1	1/1	2/2
pref_dwf	1/1	2/2	1/1	1/1	2/2
pref_dwi	1/1	2/2	1/1	1/1	2/2
pref_dwl	1/1	2/2	1/1	1/1	2/2
pref_dws	1/1	2/2	1/1	1/1	2/2
pref_dwss	1/1	2/2	1/1	1/1	1/2
pref_mmm	1/1	2/2	2/2	2/2	4/4
pref_mp	1/1	2/2	2/2	2/2	4/4
pref_mwr	1/1	2/2	2/2	2/2	3/4
pref_mww	1/1	2/2	2/2	2/2	4/4
pref_mnxm	4/4	8/8	4/4	4/4	8/8
pref_bl1	1/1	2/2	2/2	2/2	4/4
pref_bl2	1/1	2/2	1/1	2/2	4/4
pref_bl3	1/1	2/2	1/1	2/2	4/4
pref_bl4	2/2	4/4	2/2	2/2	3/4
pref_bl5	1/1	2/2	1/1	3/3	6/6
pref_bl6	1/1	0/2	0/3	0/3	-
pref_bl7	1/1	2/2	1/1	2/2	4/4
pref_bnl1	1/2	2/2	0/2	0/2	-
pref_bnl2	1/2	2/2	0/2	0/2	-
pref_bnl3	1/1	2/2	1/1	2/2	4/4

Legende: Die Buchstaben D, E und Z sind als Abkürzungen für SharedPreferences-Datei, -Eintrag und -Zugriff zu verstehen. Die Werte dieser drei Spalten beziehen sich auf ihre jeweilig pro App gefundene Anzahl gefolgt von der vorhandenen Anzahl. In der Spalte „Güte“ werden dabei jeweils erkannten Eigenschaften (Datei: Name und Modus, Eintrag/Zugriff: Schlüssel und Wert) des Aufrufes im Verhältnis zu den vorhandenen Eigenschaften angeführt.

Tabelle 4.3: Übersicht der Benchmark-Resultate

4.2 Fallstudie

Zweiter Bestandteil der Evaluation ist eine Fallstudie mit einigen veröffentlichten Android-Apps. Die ausgewählten Apps werden hierzu einer automatischen Untersuchung mit dem Analysewerkzeug unterzogen. Anschließend erfolgt zusätzlich eine manuelle Inspektion der, über etablierte Reverse-Engineering Werkzeuge wiederhergestellten, Quelltexte. Ein Vergleich der Ergebnisse dieser Verfahren und eine darauf aufbauende Bewertung der Qualität des entwickelten Analyseverfahrens erfolgen in Kapitel 5. Anschließend wird dort auch die ursprüngliche Annahme des Speicherns sensibler Daten in den SharedPreferences hinsichtlich der Befunde diskutiert.

4.2.1 Umsetzung

Die zur Untersuchung benötigten APK-Dateien müssen zuerst über den Google Play Store heruntergeladen werden. Üblicherweise passiert dies über die Play Store-App auf einem Android-Gerät in Verbindung mit einem Google-Account. Komfortabler ist dies unter Umgehung der ersten beiden Voraussetzungen über das Programm Raccoon, es ist lediglich ein Google-Account erforderlich. Dieser „Google Play Desktop Client“ [34] simuliert das Verhalten der Google Play App und erlaubt damit ein direktes Herunterladen von Apps auf ein Desktop-System. Für die manuelle Inspektion der Apps wird deren DEX-Bytecode in Java-Bytecode konvertiert, um diesen mit einem Decompiler in Java-Quelltext übersetzen zu können. Eine umfangreiche Sammlung von Konvertierungswerkzeugen zwischen .dex und .class Dateien beinhaltet dex2jar [17], aus welcher die Umwandlung von APK- in Java Archive (JAR)-Dateien über den Befehl `d2j-dex2jar` genutzt wird. Der Decompiler JD-GUI des Java Decompiler Project [39] wurde aufgrund seiner intuitiven und übersichtlichen Oberfläche ausgewählt. Für die durchgeführte Inspektion hat sich besonders die paketweite Suchfunktion als hilfreich erwiesen, da eine initiale Suche nach dem Typ SharedPreferences direkt alle interessanten Startpunkte liefert.

4.2.2 Apps

Die für die Fallstudie ausgewählten Apps, welche in Tabelle 4.4 aufgeführt sind, stammen alle aus dem Einsatzgebiet der Internet Protocol (IP)-Videoüberwachung. Die Festlegung auf eine gemeinsame Domäne bietet sich an, da die Apps so, aufgrund ähnlicher Funktionalität und möglicher Risiken, besser miteinander vergleichbar sind, als wenn sie für unterschiedliche Anwendungen konzipiert wären.

Name	Paketname	Version
ABUS IPCam	de.abussc.androidipcam.androidipcam	1.0.17
ABUS Life View	com.ideabus.abusliveview	1.0
AXIS Camera Companion	com.axis.acc	1.2.1
AXIS Wireless Install'n Tool	com.axis.wit	1.0.0
Bosch Video Security	com.bosch.onsite	1.0.1.58
GRViewer	com.grundig_cctv.grviewer	54

Anmerkung: Die App GRViewer besitzt in der vorliegenden Fassung keinen Versionsnamen (versionName), daher bezieht sich der unter Version angegebene Wert abweichend auf den Versionscode (versionCode) des Buildsystems.

Tabelle 4.4: Übersicht der ausgewählten Apps

Die untersuchten Apps ermöglichen die Verbindung zu und den Zugriff auf IP-Kameras zur Einrichtung und/oder Abruf von Bild- und/oder Videodatenstreams. Zielgruppe sind für die meisten Apps gemäß ihrer Beschreibung anscheinend Privat- und Small Office, Home Office (SOHO)-Anwender. Aus der Verwendung und dem Anwenderkreis ergeben sich mögliche Risiken der Verletzung der Privatsphäre und der Ermittlung sicherheitsgefährdender Informationen. Denkbar ist beispielsweise die Videoüberwachung einer Haustür oder eines Lagertores. Kann sich ein Angreifer Zugriff auf die Netzwerkkamera verschaffen, ermöglicht ihm dies nun zumindest die Ermittlung von An-, Abwesenheits- oder Geschäftszeiten. Im schlimmsten Fall ließe sich der Ort des unter dem Fenstersims neben der Haustür versteckten Notschlüssels oder die Zahlenkombination der Lagertür erkennen beziehungsweise ableiten. Damit sind in diesen Apps hinterlegte Zugangs- und Konfigurationsdaten besonders schützenswert und der Umgang mit diesen untersuchenswert. Weiterhin wurde bei der Auswahl darauf geachtet, dass es sich um verbreitete Produkte namhafter Hersteller handelt. Die Anbieter der untersuchten Apps bewerben deren Funktionalität wie folgt:

ABUS IPCam „ermöglicht Ihnen Livebilder von ABUS Security-Center Netzwerkkameras auf Ihrem Android Smartphone anzuzeigen. Neben den Live-Bildern können bei ausgewählten Kameras auch diverse Steuerfunktionen durchgeführt werden.“ [22]

ABUS Life View „ermöglicht Ihnen den einfachen und schnellen Zugriff auf Ihre VGA Netzwerk Kompaktkamera [...]. Live-Bilder können per WiFi oder direkt über das

Internet übertragen werden.“ [23]

AXIS Camera Companion - „a simple, yet sophisticated solution for video surveillance with typically 1-4 cameras and with support for up to 16 cameras. The viewing app utilizes the powerful capabilities and smart features of Axis IP-cameras.“ [24]

AXIS Wireless Install'n Tool „for the Axis T8415 Wireless installation tool. Simplicity in your hand. Install Axis cameras with ease.“ [25]

Bosch Video Security verbindet „sich von überall auf der Welt mit Ihren Bosch H.264-IP-Kameras und -Encodern“ und bietet „unmittelbare Videowiedergabe, vollständigen Zugriff auf Ihre Aufzeichnungen, forensische Suche auf Kameras mit Bosch Videoinhaltsanalyse-Unterstützung (IVA) und flüssige Steuerung [...].“ [26]

GRViewer (Grundig) bietet "Mehrkanal Live Ansicht, unterschiedliche Such- und Wiedergabefunktionen, Zugriff auf die Konfiguration Ihrer Rekorder."[27]

4.2.3 Ergebnisse

ABUS IPCam

Für diese App findet die automatische Analyse, wie aus Listing 4.2) ersichtlich, eine unbekannte, eine als `de.abussc.androidipcam.androidipcam_preferences.xml` identifizierte und eine als `preferences.xml` identifizierte SharedPreferences-Datei, welche jeweils mit der `MODEPRIVATE`-Berechtigung verwendet werden. Der einzige erkannte Eintrag ist der Boolean `pin_enabled` mit zwei zugreifenden Methoden. Der Defaultwert für den Lesezugriff in `isRememberPin` ist 0 (`False`). Der Wert für den Schreibzugriff in `saveRemem⌋ berPin` konnte nicht bestimmt werden. Erkennbar ist dies an der ausgegebenen Soot-Variablenbezeichnung `$z0`. Dem scheinbar enthaltenen Begriff PIN nach steht dieser Eintrag möglicherweise in Zusammenhang mit sicherheitsrelevanten Informationen, da PINs üblicherweise als numerische Passwörter Verwendung finden.

```
APP: de.abussc.androidipcam.androidipcam
UNKNOWN (MODEPRIVATE)
de.abussc.androidipcam.androidipcam_preferences.xml (MODEPRIVATE)
> pin_enabled - BOOLEAN
    isRememberPin.READ(0)
    saveRememberPin.WRITE($z0)
preferences.xml (MODEPRIVATE)
```

Listing 4.2: Ergebnis der automatischen Analyse (ABUS IPCam)

Die manuelle Inspektion bestätigt die automatisch gefundenen Zugriffe und Werte. Listing 4.3 zeigt die Verwendung der Datei `de.abussc.androidipcam.androidipcam_preferences.xml` in der Klasse `PinWatcher` aus dem Paket `de.abussc.androidipcam.pin`. Allerdings ist diese Verwendung nicht als problematisch einzustufen, da der Eintrag `pin_enabled` als Hilfsvariable für den Status einer Konfguration oder Aktion dient.

```
76   public boolean isRememberPin()
77   {
78     return PreferenceManager.getDefaultSharedPreferences((Context)this.contex
     ↪  t.get()).getBoolean("pin_enabled",
     ↪  false);
79   }
```

```
86   public void saveRememberPin(boolean paramBoolean)
87   {
88     SharedPreferences.Editor localEditor = PreferenceManager.getDefaultShared
     ↪  Preferences((Context)this.context.get()).edit();
89     localEditor.putBoolean("pin_enabled", paramBoolean);
90     localEditor.commit();
91   }
```

Listing 4.3: Verwendung von SharedPreferences in `PinWatcher`

Zusätzlich werden SharedPreferences-Objekte in den Klassen `Protectionmanager` (gleiches Paket), `SupplementContext` (`de.abussc.androidipcam`) und `SecurePreferences` (`de.abussc.androidipcam.utils`) referenziert. Die in `Protectionmanager` enthaltenen Methoden `isPinRemembered` und `rememberPin` sind dabei wieder Getter- und Setter-Methoden auf dem Eintrag `pin_enabled` und damit funktional identisch mit der in `PinWatcher` ent-

haltenen Funktionalität (Listing 4.3). `SupplementContext` implementiert ebenfalls eine, zu der in `PinWatcher` identische, `saveRememberPin`. Die Klasse `SecurePreferences` erweitert ihrerseits Zugriffe auf SharedPreferences-Objekte um Ver- und Entschlüsselung von Einträgen und Schlüsseln. Ihre Verwendung erfolgt innerhalb der App über die Klasse `SecurePreferencesManager`. Zur Veranschaulichung stellen Listing 4.4 bis 4.6 die Kette von Aufrufen dar, die in `ServerSetup` im Paket `de.abussc.androidipcam.wizard.ui` bei der Sicherung von Zugangsdaten erfolgen. Die eigentliche Ver- und Entschlüsselung ist per Advanced Encryption Standard (AES) über Java-Standardbibliotheken aus `javax.crypto` und `javax.security` implementiert. Bemerkenswerterweise erfolgt die in Listing 4.7 dargestellte Initialiserung der `SecurePreferences` mit einem statischen Schlüssel. Die in Listing 4.8 und 4.9 dargstellte Verwendung zum Aufsetzen der Verschlüsselung verbunden mit der Feststellung, dass die in Zeile 155 damit aufgerufene Methode `getSecretKey` ihrerseits auch keine Zufallskomponenten in den Schlüssel einbringt, gibt weiteren Grund zur Zweifel an der Wirksamkeit der Verschlüsselung.

```
205   private void saveAbusServerCredentials()
206   {
207     this.securePreferencesManager.saveValue("key_abus_user", this.abusUser);
208     this.securePreferencesManager.saveValue("key_abus_password",
      ↪  this.abusPassword);
209   }
```

Listing 4.4: Speichern von Zugangsdaten (`ServerSetup`)

```
73   public void saveValue(String paramString1, String paramString2)
74   {
75     this.securePreferences.put(paramString1, paramString2);
76   }
```

Listing 4.5: Methode zur Verschlüsselung von Werten (`SecurePreferencesManager`)

```
66  private void putValue(String paramString1, String paramString2)
67    throws SecurePreferences.SecurePreferencesException
68  {
69    paramString2 = encrypt(paramString2, this.writer);
70    this.preferences.edit().putString(paramString1, paramString2).commit();
71  }
```

Listing 4.6: Methode zur Verschlüsselung von Werten (SecurePreferences)

```
15  private static String SECURE_KEY = "28fhFJfb20ch2k";
```

```
20  public SecurePreferencesManager(Context paramContext)
21  {
22    this.context = paramContext;
23    this.securePreferences = new SecurePreferences(paramContext,
    ↪  PREFERENCES_NAME, SECURE_KEY, true);
24  }
```

Listing 4.7: Initialisierung der SecurePreferences (SecurePreferencesManager)

```
29  public SecurePreferences(Context paramContext, String paramString1, String
    ↪  paramString2, boolean paramBoolean)
30    throws SecurePreferences.SecurePreferencesException
31  {
32    try
33    {
34      this.writer = Cipher.getInstance("AES/CBC/PKCS5Padding");
35      this.reader = Cipher.getInstance("AES/CBC/PKCS5Padding");
36      this.keyWriter = Cipher.getInstance("AES/ECB/PKCS5Padding");
37      initCiphers(paramString2);
38      this.preferences = paramContext.getSharedPreferences(paramString1, 0);
39      this.encryptKeys = paramBoolean;
40      return;
41    }
```

Listing 4.8: Initialisierung der Verschlüsselungsverfahren I (SecurePreferences)

```
151   protected void initCiphers(String paramString)
152     throws UnsupportedEncodingException, NoSuchAlgorithmException,
    ↪  InvalidKeyException, InvalidAlgorithmParameterException
153   {
154     IvParameterSpec localIvParameterSpec = getIv();
155     paramString = getSecretKey(paramString);
156     this.writer.init(1, paramString, localIvParameterSpec);
157     this.reader.init(2, paramString, localIvParameterSpec);
158     this.keyWriter.init(1, paramString);
159   }
```

Listing 4.9: Initialisierung der Verschlüsselungsverfahren II (`SecurePreferences`)

ABUS Life View

Weder automatische Analyse (vergleiche Listing 4.10) noch manuelle Inspektion geben bei dieser App Hinweise darauf, dass SharedPreferences verwendet werden.

```
1 APP: com.ideabus.abusliveview
```

Listing 4.10: Ergebnis der automatischen Analyse (ABUS Life View)

AXIS Camera Companion

Die automatische Analyse liefert als Ergebnis zu dieser App lediglich eine unbekannte SharedPreferences-Datei mit der `MODEPRIVATE`-Berechtigung, wie aus Listing 4.11 zu erkennen ist.

```
1 APP: com.axis.acc
2 UNKNOWN (MODEPRIVATE)
```

Listing 4.11: Ergebnis der automatischen Analyse (AXIS Camera Companion)

Durch manuelle Inspektion wird ersichtlich, dass tatsächlich eine Verwendung von SharedPreferences stattfindet. Über drei Klassen des Paketes `com.axis.acc.helpers` werden Versionsinformationen (`AppPrefsHelper`), Einstellungen zur Videoauflösung (`SettingsPr⌋ efsHelper`) und Zugangsdaten (`AuthPrefsHelper`) verwaltet. Die Klassen `AppPrefsHelper` und `SettingsPrefsHelper` erweitern dazu `PrefsHelper` aus dem Paket `com.axis.lib.ut⌋`

il. Die Klasse `AuthPrefsHelper` erweitert hingegen die ebenfalls dort verortete `CryptoPrefsHelper` zur Ver- und Entschlüsselung von Werten in SharedPreferences-Einträgen, welche ihrerseits auch wieder auf `PrefsHelper` aufsetzt. Die tatsächliche Implementierung der Ver- und Entschlüsselungsmethoden ist in `CryptoHelper` enthalten. Zum Beispiel erfolgt eine Abfrage des Passwortes zu einer gespeicherten Verbindung über den Aufruf von `getSitePassword` mit dem Parameter der Site-ID in `AuthPrefsHelper`. Wie Listing 4.12 zeigt, wird diese ID, mit dem Präfix *`"password"`* versehen, an die Methode `getEncrypted` aus `CryptoPrefsHelper` übergeben. Dort (Listing 4.13) erfolgt der Abruf des verschlüsselten Eintrages, welcher in den SharedPreferences `prefs` unter dem als Parameter übergebenen zusammengesetzten Schlüssel abgelegt ist. Mit dem lokalen String `url` wird sie an den Aufruf von `decrypt` in `CryptoHelper` (siehe Listing 4.14, Zeile 19 bis 25) weitergegeben. Aus `url` wird dort in der Methode `getRawKey` (ab Zeile 73) ein AES-Schlüssel erzeugt, mit welchem anschließend eine Entschlüsselung (Zeile 37 bis 44) durchgeführt wird. Das zur Ver- und Entschlüsselung eingesetzte AES-Verfahren wird über Java-Standardbibliotheken aus `javax.crypto` und `javax.security` umgesetzt.

```
50  public String getSitePassword(String paramString)
51  {
52    return getEncrypted("password" + paramString);
53  }
```

Listing 4.12: Getter-Methode für das Passwort einer Site (`AuthPrefsHelper`)

```
18  protected String getEncrypted(String paramString)
19  {
20    paramString = this.prefs.getString(paramString, "");
21    return CryptoHelper.decrypt(this.url, paramString);
22  }
```

Listing 4.13: Methode zur Entschlüsselung von Einträgen (`CryptoPrefsHelper`)

```
19  public static String decrypt(String paramString1, String paramString2)
20  {
21    byte[] arrayOfByte = new byte[0];
22    try
23    {
24      paramString1 = decrypt(getRawKey(paramString1.getBytes()),
    ↪ ByteUtils.toByte(paramString2));
25      return new String(paramString1);
```

```
37  private static byte[] decrypt(byte[] paramArrayOfByte1, byte[]
    ↪ paramArrayOfByte2)
38    throws Exception
39  {
40    paramArrayOfByte1 = new SecretKeySpec(paramArrayOfByte1, "AES");
41    Cipher localCipher = Cipher.getInstance("AES");
42    localCipher.init(2, paramArrayOfByte1);
43    return localCipher.doFinal(paramArrayOfByte2);
44  }
```

```
73  private static byte[] getRawKey(byte[] paramArrayOfByte)
74    throws Exception
75  {
76    KeyGenerator localKeyGenerator = KeyGenerator.getInstance("AES");
77    try
78    {
79      SecureRandom localSecureRandom1 = SecureRandom.getInstance("SHA1PRNG",
    ↪ "Crypto");
80      localSecureRandom1.setSeed(paramArrayOfByte);
81      localKeyGenerator.init(128, localSecureRandom1);
82      return localKeyGenerator.generateKey().getEncoded();
83    }
```

Listing 4.14: Entschlüsselung und Schlüsselerzeugung (`CryptoHelper`)

AXIS Wireless Install'n Tool

Bei dieser App liefert die automatische Analyse eine unbekannte und eine als `genera⌋ l_preferences.xml` identifizierte SharedPreferences-Datei, wobei beide mit der `MODEPR⌋ IVATE`-Berechtigung verwendet werden. Die Ausgabe des Ergebnisses ist in Listing 4.15

wiedergegeben. Als einziger Eintrag in `general_preferences.xml` wurde der Boolean `pie_menu_hint_discarded` mit zwei zugreifenden Methoden erkannt. Der Defaultwert für den Lesezugriff in `hasDiscardedPieMenuHint` ist 0 (`False`). Der Wert für den Schreibzugriff in `setPieMenuHintDiscarded` ist 1 (`True`). Den Begriffen nach handelt es sich hierbei offenbar um eine Nutzer-Präferenz zum Ausblenden eines Hinweises (hint) zu einem GUI-Element, dem PieMenu.

```
APP: com.axis.wit
UNKNOWN (MODEPRIVATE)
general_preferences.xml (MODEPRIVATE)
> pie_menu_hint_discarded - BOOLEAN
    hasDiscardedPieMenuHint.READ(0)
    setPieMenuHintDiscarded.WRITE(1)
```

Listing 4.15: Ergebnis der automatischen Analyse (AXIS Wireless Install'n Tool)

Über die manuelle Inspektion lässt sich dieser Befund bestätigen. Die bereits erkannte Verwendung der Datei `general_preferences.xml` in der Klasse `VideoActivity` belegt Listing 4.16.

```
179   private boolean hasDiscardedPieMenuHint()
180   {
181     return getSharedPreferences("general_preferences",
      ↪  0).getBoolean("pie_menu_hint_discarded", false);
182   }
```

```
254   private void setPieMenuHintDiscarded()
255   {
256     SharedPreferences.Editor localEditor =
      ↪  getSharedPreferences("general_preferences", 0).edit();
257     localEditor.putBoolean("pie_menu_hint_discarded", true);
258     localEditor.commit();
259   }
```

Listing 4.16: Verwendung von SharedPreferences in `VideoActivity`

Darüber hinaus kommt es zur weiteren Verwendung von SharedPreferences wie in der zuvor untersuchten AXIS Camera Companion App. Die im Paket `com.axis.wit.helpers` enthaltende Klasse `CredentialsPrefsHelper` verwaltet Zugangsdaten und greift dazu auf

die bekannten Klassen `CryptoPrefsHelper`, `CryptoHelper` und `PrefsHelper` aus dem Paket `com.axis.lib.util` zurück.

Bosch Video Security

Leider erwies sich die automatische Untersuchung für diese App als nicht durchführbar, da die Soot-Analyse bereits vor erreichen des SharedPreferences-Plugins abstürzte. Auslöser ist eine `java.lang.ClassCastException`, welche in der Phase der Jimple-Umwandlung (`d.u.s.s.p.t.JimpleTransformer`) auftritt. Durch die manuelle Inspektion lassen sich drei Klassen mit SharedPreferences-Verwendung feststellen. Dabei handelt es sich um die im Paket `com.bosch.onsite.app` beinhalteten `LoginActivity`, `SiteEditActivity` und `SiteSelectionActivity`. Die Methodennamen `loadCredentials` und `storeCredentials`, je in `LoginActivity` und `SiteEditActivity` enthalten, sowie `removeCredentials`, aus `SiteSelectionActivity`, lassen bereits auf eine Verwendung zur Verwaltung von Zugangsdaten schließen. Listing 4.17 zeigt die erstgenannten Methoden aus der Klasse `SiteEditActivity`. Die Implementierungen in `LoginActivity` sind nahezu identisch und auch `removeCredentials` aus `SiteSelectionActivity` folgt diesem Muster. Offensichtlicherweise werden Benutzername und Passwort jeweils im Klartext aus SharedPreferences geladen und in diese gespeichert. Auch in den anderen Klassen, welche unter anderem die zugehörige GUI verwalten, erfolgt keine feststellbare Verschlüsselung. Darüber hinaus werden die Zugangsdaten beim Laden potentiell auch in Logeinträgen ausgegeben.

```
100   void loadCredentials()
101   {
102     SharedPreferences localSharedPreferences = getSharedPreferences(".auth",
      ↪  0);
103     String str = this.mLocalLobby.uuid.toString();
104     this.mUsername = localSharedPreferences.getString(str + "_usr", "");
105     this.mPassword = localSharedPreferences.getString(str + "_pwd", "");
106     this.mCredentialsChanged = false;
107     Log.d("SiteEditActivity", "Loading credentials for " + str + " got: " +
      ↪  this.mUsername + "," + this.mPassword);
108   }
```

```
263   void storeCredentials()
264   {
265     if (this.mLocalLobby != null)
266     {
267       SharedPreferences.Editor localEditor = getSharedPreferences(".auth",
      ↪  0).edit();
268       String str = this.mLocalLobby.uuid.toString();
269       Log.d("SiteEditActivity", "Storing credentials for " + str);
270       localEditor.putString(str + "_usr", this.mUsername);
271       localEditor.putString(str + "_pwd", this.mPassword);
272       localEditor.commit();
273       return;
274     }
275     Log.d("SiteEditActivity", "Not storing credentials - no LobbyFile yet.");
276   }
```

Listing 4.17: Verwendung von SharedPreferences in `SiteEditActivity`

GRViewer (Grundig)

Leider scheiterte auch für diese App die automatische Untersuchung, da die Soot-Analyse bereits vor erreichen des SharedPreferences-Plugins abstürzte. Der aufgetretene Fehler (`java.lang.ClassCastException`) entspricht dem der zuvor behandelten Bosch Video Security App. Die manuelle Inspektion zeigt keine Verwendung von SharedPreferences mit kritischen Informationen. Einige in dem Paket `com.nviewer.dvrviewer.activities` enthaltenen Klassen verwenden diese offenbar zur Speicherung von Einstellungen hinsichtlich der Bildqualität. Beispielhaft dafür zeigt Listing 4.18 einen Auszug aus der Klasse

NViewerSetupActivity.

```
45   protected void onCreate(Bundle paramBundle)
46   {
47     super.onCreate(paramBundle);
48     setContentView(R.layout.nviewer_setup_layout);
49     this.context = this;
50     this.mDBAdapter = DBAdapter.getInstance(this);
51     this.hdCheck = ((CheckBox)findViewById(R.id.check_nviewer_setup_hd));
52     this.prefs = getSharedPreferences("PrefName", 0);
53     this.edit = this.prefs.edit();
54     if (this.prefs.getString("HD", "NOTHD").equals("HD")) {
55       this.hdCheck.setChecked(true);
56     }
```

Listing 4.18: Verwendung von SharedPreferences in NViewerSetupActivity

5 Diskussion der Ergebnisse

Die Bewertung der Qualität des in dieser Arbeit entwickelte Analyseverfahrens anhand der Ergebnisse von Benchmarktest und Fallstudie soll Ausgangspunkt der Diskussion der Ergebnisse sein. Die in der Fallstudie gewonnenen Erkentnisse hinsichtlich der Verwendung von SharedPreferences in Android-Apps dienen danach als Grundlage einer Bewertung der ursprünglich angenommenen Hypothese.

5.1 Qualität der automatischen Analyse

Bei der Durchführung des Benchmarks konnten hohe Precision- und Recall-Werte von 0,95 und 0,93 erzielt werden. Bei Nichtberücksichtigung der einfacher konzipierten Benchmarkfällen bleiben die Werte immerhin noch bei einer Precision von 0,86 und einem Recall von 0,82. Rein nach diesen Kennzahlen ließe sich die vorgenommene Implementierung also durchaus als erfolgreich bezeichnen. Zur Relativierung müssen allerdings an dieser Stelle zwei Problematiken hinsichtlich des Evaluationsverfahren erwähnt werden. Die erste besteht darin, dass die Entwicklung der Benchmark-Apps und des Analyseverfahrens beide durch den Autor dieser Arbeit, und damit aus einer Hand, erfolgten und damit Untersucher-Bias angenommen werden muss. Selbst wenn es gelungen sein sollte, die Kenntnis der Implementierung der jeweils anderen Komponente auszublenden, so bestehen doch gemeinsame Annahmen, Denkmuster und Präferenzen, die weiter zum Tragen kämen. Idealerweise hätte zur Evaluation eine standardisierte Benchmark-Suite zur Verfügung gestanden oder die Benchmark-Fälle wären durch unabhängige Dritte spezifiziert und entwickelt worden. Das zweite Problem besteht in der damit verbundenen Abhängigkeit der Precision und Recall-Metriken von der Qualität und Quantität der vorhanden Benchmark-Fälle. Aus diesem Grund hielt der Autor auch eine zusätzliche Betrachtung ohne Aussparung der einfachen Fälle für sinnvoll. Zudem ist festzustellen, dass die vereinzelt nicht erkannten Werte für Parameter bereits Schwächen des Verfahrens hinsichtlich des Umgangs mit komplexeren Datenstrukturen und -flüssen erkennen lassen. Bei der Durchführung der automatischen Analyse auf Apps aus dem Google Play Store ließen sich

für vier der sechs untersuchten Apps Ergebnisse ermitteln. Bei diesen kam es in drei von vier Fällen zur unvollständigen Erkennung von SharePreferences-Zugriffen. Für die betreffenden Apps (ABUS IPCam, AXIS Camera Companion und AXIS Wireless Install'n Tool) wurde die Instanzierung von SharePreferences-Objekten zwar erkannt, allerdings wurden weder die vollständigen Eigenschaften (Name, Modus) noch Zugriffe darauf erkannt. Grund dafür war immer die nicht-lokale, methodenübergreifende Verwendung von SharedPreferences, welche bei Implementierung des Analyseverfahrens nicht vorgesehen war. Eine lokale Verwendung von SharedPreferences konnte durch die automatische Untersuchung bei zwei Apps (ABUS IPCam und AXIS Wireless Install'n Tool) festgestellt werden. Die vierte automatisch untersuchte App verwendet keine SharedPreferences, was auch korrekt ausgegeben wurde.

5.2 Verwendung von SharedPreferences

In Bezug auf die Annahme, dass in Android-Apps möglicherweise sensible Informationen in SharedPreferences abgelegt werden, lassen sich mit der in der Fallstudie ergänzend vorgenommenen manuellen Inspektion von dekompiliertem Quelltext folgende Angaben machen: Fünf der sechs untersuchten Apps nutzen SharedPreferences zur Speicherung von Schlüssel-Wert Paaren. Immerhin noch vier der Apps, und zwar ABUS IPCam, AXIS Camera Companion, AXIS Wireless Install'n Tool und Bosch Video Security, legen darin auch sensible Informationen in Form von Zugangsdaten für Videoüberwachungssysteme ab. Drei dieser vier Apps kapseln den Zugriff auf schützenwerte Informationen mit Ver- und Entschlüsselung über gängigen Java-Bibliotheken. Ohne eine Validierung der Implementierung der Verschlüsselungsverfahren vorzunehmen, scheint diese bei den beiden AXIS-Apps schlüssig. Die Entscheidung den eingesetzten Schlüssel, im Falle der ABUS IPCam App, im Quelltext (hardcoded) zu hinterlegen ist dahingegen als problematisch zu werten. Übertroffen wird dies nur in der Bosch Video Security App, welche Zugangsdaten unverschlüsselt über die SharedPreferences verwaltet. Natürlich verbietet es der stichprobenartige Charakter der vorgenommenen Untersuchung allgemeingültige Aussagen zu treffen. Allerdings bleibt anhand der vorliegenden Indizien festzustellen, dass es unter professionellen Entwicklern, auch in einer Domäne mit einem solch unmittelbarem Sicherheitsbezug wie der IP-Videoüberwachung, vereinzelt ein zu hohes Vertrauen auf Android-Sicherheitsmechanismen (Sandboxing) oder eine mangelnde Sensibilität im Umgang mit schützenswerten Daten zu geben scheint.

6 Fazit und Ausblick

Das Ziel der vorliegenden Arbeit war es, die Verwendung von SharedPreferences in Android-Apps mit Hilfe statischer Programmanalyse zu untersuchen. Von besonderem Interesse war dabei die mögliche Speicherung von sensiblen Daten in diesen. Vor dem Hintergrund der potenziellen Umgehung bestehender Android-Sicherheitsmechanismen über Sicherheitslücken wäre diese als Sicherheitsrisiko einzustufen, da einem Angreifer über das Erlangen von Berechtigungen ein Zugriff auf die in XML-Dateien hinterlegten Informationen gelingen könnte. Dazu wurde auf Basis des ZertApps-Demonstrators ein Soot-Plugin entwickelt, welches intraprozedurale Zugriffe auf SharedPreferences über die Auswertung von Jimple-Ausdrücken erkennt. Ergebnisse werden an das ZertApps-Modell angehängt und können in Verbindung mit den Ergebnissen anderer Analysen als Grundlage für eine Einschätzung des Verhaltens der untersuchten App unter sicherheitsrelevanten Gesichtspunkten dienen.

Die Evaluation des implementierten Analyseverfahrens erfolgte über einen eigens konzipierten Bechmarking-Test und die Durchführung einer Fallstudie mit Apps aus dem Google Play Store. Dabei konnte die Funktion des Soot-Plugins, mit Einschränkungen bei der Auswertung komplexerer Datenstrukturen und der generellen Robustheit, demonstriert werden. In der vorliegenden Form sind seine Ergebnisse zum Beispiel als Indikator für eine erweiterte Analyse der SharedPreferences-Verwendung einer App nutzbar. Für eine zukünftige Erweiterung wären verschiedene Ansätze vorstellbar. Neben einer Adressierung der genannten Einschränkungen hinsichtlich komplexer Datenstrukturen durch bessere Auflösung der Jimple-Variablen über eine Def-Use Datenflussanalyse bietet sich vor allem auch eine Erweiterung der Analyse um eine Erkennung von interprozeduralen Aufrufen und Verweisen an. Zusätzlich könnte eine Änderung der Modellierung und der Implementierung bezüglich des Verhältnisses zwischen gefundenen Dateien und Zugriffen sinnvoll sein. Dadurch würden erkannte Zugriffe, die keiner Datei zuzuordnen sind, zusätzlich in die Ergebnisse aufgenommen werden. Da die angeführten Erweiterungen aber erwartungsgemäß mit höheren Kosten für die Analyse verbunden sind, müssten diese gegen die zu erreichende Genauigkeit abgewägt werden.

In Bezug auf die Verwendung von SharedPreferences lässt sich anhand der Ergebnisse der Fallstudie feststellen, dass die Eingangshypothese für die getroffene Stichprobe bestätigt werden konnte. Dass es bei den untersuchten Apps im Umgang mit Zugangsdaten zu sensiblen Systemen, neben offenbar ineffektiver Verschlüsselung bei einem Fall, sowie ein Lesen und Speichern im Klartext bei einem anderen Fall, kommt, ist als problematisch zu werten. Nur eine korrekt implementierte Verschlüsselung gewährleistet, dass in SharedPreferences gespeicherte Daten im Falle eines Angriffes auf das Gerät mit Folge der Erlangung von Zugriffsberechtigungen auf diese, also einer Umgehung des Sandboxing-Prinzips in Android, nicht lesbar sind. Interessant wäre zukünftig die Untersuchung der Fragestellung dieser Arbeit auch in Form einer größer angelegten, quantitativen Studie, um allgemeingültigere Aussagen treffen zu können.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

AES Advanced Encryption Standard

API Application Programming Interface

APK Android Package

ART Android Runtime

AST Abstract Syntax Tree

CFG Control Flow Graph

DAC Discretionary Access Control

DEX Dalvik Executable

DFD Data Flow Diagram

EMF Eclipse Modelling Framework

GID Group Identifier

GUI Graphical User Interface

ID Identifier

IDE Integrated Development Environment

IP Internet Protocol

IPC Inter Process Communication

HAL Hardware Abstraction Layer

IR Intermediate Representation

JAR Java Archive

JNI Java Native Interface

JVM Java Virtual Machine

MMS Multimedia Messaging Service

PID Process Identifier

PIN Personal Identification Number

POSIX Portable Operating System Interface

RPC Remote Procedure Call

SDK Software Development Kit

SMS Short Message Service

SOHO Small Office, Home Office

SSA Static Single Assignment

TAC Three Address Code

UID User Identifier

UML Unified Modeling Language

VM Virtual Machine

XML Extensible Markup Language

Literaturverzeichnis

[1] Alfred V. Aho, Hrsg. *Compilers: principles, techniques, & tools*. 2nd ed. Boston: Pearson/Addison Wesley, 2007.

[16] Steffen Bartsch u. a. „Zertifizierte Datensicherheit für Android-Anwendungen auf Basis statischer Programmanalysen“. In: *Sicherheit 2014 Sicherheit, Schutz und Zuverlässigkeit*. GI SICHERHEIT 2014 Sicherheit – Schutz und Zuverlässigkeit. Hrsg. von Volkmar Stefan und Edgar Weippl. Bd. 228. LNI. Wien: GI, 2014, S. 283–291. URL: `http://www.informatik.uni-bremen.de/~sohr/papers/GI14.pdf` (besucht am 26. 12. 2015).

[18] Dan Bornstein. „Dalvik VM Internals“. Google I/O 2008. San Francisco, CA, 2008. URL: `https://sites.google.com/site/io/dalvik-vm-internals` (besucht am 27. 12. 2015).

[19] Árni Einarsson und Janus Dam Nielsen. *A Survivor's Guide to Java Program Analysis with Soot*. 17. Juli 2008. URL: `http://www.brics.dk/SootGuide/` (besucht am 11. 12. 2015).

[20] Nikolay Elenkov. *Android security internals: an in-depth guide to android's security architecture*. 1st edition. San Francisco, CA: No Starch Press, 2014.

[29] Rainer Koschke. „Software-Reengineering (03-05-H-706.01)“. Vorlesungsunterlagen. Bremen, 2013. URL: `http://www.informatik.uni-bremen.de/st/lehredetails.php?id=302&lehre_id=313` (besucht am 13. 01. 2016).

[30] Patrick Lam u. a. „The Soot framework for Java program analysis: a retrospective“. In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011). Okt. 2011. URL: `http://www.bodden.de/pubs/lblh11soot.pdf` (besucht am 22. 01. 2016).

[31] Li Li u. a. „IccTA: Detecting inter-component privacy leaks in Android apps“. In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, S. 280–291. URL: http://dl.acm.org/citation.cfm?id=2818791 (besucht am 21. 03. 2016).

[35] Bradley Reaves u. a. „Mo(Bile) Money, Mo(Bile) Problems: Analysis of Branchless Banking Applications in the Developing World“. In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, S. 17–32. URL: http://dl.acm.org/citation.cfm?id=2831143.2831145 (besucht am 15. 02. 2016).

[38] Andrew S. Tanenbaum und Herbert Bos. *Modern operating systems*. Global ed., 4. ed. Always learning. Boston: Pearson, 2015. 1101 S.

[42] Yury Zhauniarovich. *Android Security (and Not) Internals*. Web, 2014. URL: http://zhauniarovich.com/files/asani/asani.pdf (besucht am 11. 12. 2015).

Webseiten

[2] Android Open Source Project. *Android 5.0 Behavior Changes | Android Developers*. 2014. URL: https://developer.android.com/about/versions/android-5.0-changes.html#ART (besucht am 24. 01. 2016).

[3] Android Open Source Project. *Android Studio Overview | Android Developers*. URL: http://developer.android.com/tools/studio/index.html (besucht am 09. 02. 2016).

[4] Android Open Source Project. *Application security*. URL: http://source.android.com/security/overview/app-security.html (besucht am 19. 02. 2016).

[5] Android Open Source Project. *Configuring Gradle Builds | Android Developers*. URL: http://developer.android.com/tools/building/configuring-gradle.html (besucht am 09. 02. 2016).

[6] Android Open Source Project. *Download Android Studio and SDK Tools | Android Developers*. URL: http://developer.android.com/sdk/index.html (besucht am 09. 02. 2016).

[7] Android Open Source Project. *Nexus Security Bulletin - December 2015*. URL: http://source.android.com/security/bulletin/2015-12-01.html (besucht am 31. 01. 2016).

[8] Android Open Source Project. *Nexus Security Bulletin - January 2016*. URL: http://source.android.com/security/bulletin/2016-01-01.html (besucht am 31. 01. 2016).

[9] Android Open Source Project. *Nexus Security Bulletin - November 2015*. URL: http://source.android.com/security/bulletin/2015-11-01.html (besucht am 31. 01. 2016).

[10] Android Open Source Project. *Saving Key-Value Sets | Android Developers*. URL: http://developer.android.com/training/basics/data-storage/shared-preferences.html (besucht am 03. 02. 2015).

[11] Android Open Source Project. *SharedPreferences | Android Developers*. URL: http://developer.android.com/reference/android/content/SharedPreferences.html (besucht am 03. 02. 2016).

[12] Android Open Source Project. *SharedPreferences.Editor | Android Developers*. URL: http://developer.android.com/reference/android/content/SharedPreferences.Editor.html (besucht am 03. 02. 2016).

[13] Android Open Source Project. *Storage Options | Android Developers*. URL: http://developer.android.com/guide/topics/data/data-storage.html#pref (besucht am 03. 02. 2016).

[14] Android Open Source Project. *System and kernel security*. URL: http://source.android.com/security/overview/kernel-security.html (besucht am 19. 02. 2016).

[15] Android Open Source Project. *<uses-sdk> | Android Developers*. URL: http://developer.android.com/guide/topics/manifest/uses-sdk-element.html (besucht am 05. 02. 2016).

[17] Bob Pan. *pxb1988/dex2jar*. GitHub. URL: https://github.com/pxb1988/dex2jar (besucht am 17. 02. 2016).

[21] Gartner Inc. *Gartner Says Smartphone Sales Surpassed One Billion Units in 2014*. 3. März 2015. URL: http://www.gartner.com/newsroom/id/2996817 (besucht am 28. 01. 2016).

[22] Google Inc. *ABUS IPCam – Android-Apps auf Google Play*. URL: https://play.google.com/store/apps/details?id=de.abussc.androidipcam.androidipcam (besucht am 04. 02. 2016).

[23] Google Inc. *ABUS Life View – Android-Apps auf Google Play*. URL: https://play.google.com/store/apps/details?id=com.ideabus.abusliveview (besucht am 04. 02. 2016).

[24] Google Inc. *AXIS Camera Companion – Android-Apps auf Google Play.* URL: https://play.google.com/store/apps/details?id=com.axis.acc (besucht am 04. 02. 2016).

[25] Google Inc. *AXIS Wireless Install'n Tool – Android-Apps auf Google Play.* URL: https://play.google.com/store/apps/details?id=com.axis.wit (besucht am 04. 02. 2016).

[26] Google Inc. *Bosch Video Security – Android-Apps auf Google Play.* URL: https://play.google.com/store/apps/details?id=com.bosch.onsite (besucht am 04. 02. 2016).

[27] Google Inc. *GRViewer – Android-Apps auf Google Play.* URL: https://play.google.com/store/apps/details?id=com.grundig_cctv.grviewer (besucht am 04. 02. 2016).

[28] International Data Corporation (IDC). *Android and iOS Squeeze the Competition, Swelling to 96.3% of the Smartphone Operating System Market for Both 4Q14 and CY14, According to IDC.* 24. Feb. 2015. URL: http://www.idc.com/getdoc.jsp?containerId=prUS25450615 (besucht am 01. 03. 2016).

[32] Neil Mawston. *Android Shipped 1 Billion Smartphones Worldwide in 2014.* 29. Jan. 2015. URL: https://www.strategyanalytics.com/strategy-analytics/blogs/devices/smartphones/smart-phones/2015/03/11/android-shipped-1-billion-smartphones-worldwide-in-2014#.VZPpIPntlBd (besucht am 28. 01. 2016).

[33] Oracle Corporation. *Java Platform Standard Edition 8 Documentation.* URL: https://docs.oracle.com/javase/8/docs/ (besucht am 15. 02. 2016).

[34] Patrick Ahlbrecht. *Raccoon - APK downloader | Onyxbits.* URL: http://www.onyxbits.de/raccoon (besucht am 17. 02. 2016).

[36] Sable Research Group. *Soot - A framework for analyzing and transforming Java and Android Applications.* URL: http://sable.github.io/soot/ (besucht am 03. 02. 2016).

[37] Statista - The Statistics Portal. *Number of available applications in the Google Play Store from December 2009 to November 2015.* Statista. URL: http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ (besucht am 02. 03. 2016).

[39] The Java Decompiler project. *Java Decompiler.* URL: http://jd.benow.ca/ (besucht am 17. 02. 2016).

[40] ZertApps-Konsortium. *Zertapps » Aktuelles*. URL: http://www.zertapps.de/aktuelles/ (besucht am 21. 12. 2015).

[41] ZertApps-Konsortium. *Zertapps » Projekt*. URL: http://www.zertapps.de/projekt/ (besucht am 21. 12. 2016).

[43] Zimperium, Inc. *Experts Found a Unicorn in the Heart of Android*. URL: http://blog.zimperium.com/experts-found-a-unicorn-in-the-heart-of-android/ (besucht am 10. 02. 2016).

Anhang

Dieser Arbeit liegt ein Anhang in Form eines optischen Datenträgers (CD/DVD) bei. Neben Quelltexten zu der Analyse und den Benchmarkapps beinhaltet dieser auch eine digitale Fassung der Arbeit und weiteres, ergänzendes Material. Eine genaue Auflistung der Inhalte findet sich als README-Textdatei in dessen Hauptverzeichnis.