Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Tobias Markmann

Eine Android-Erweiterung zur Einschränkung von Rechteausweitung mittels dynamischer Darstellung von Informationsflüssen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Tobias Markmann

Eine Android-Erweiterung zur Einschränkung von Rechteausweitung mittels dynamischer Darstellung von Informationsflüssen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. habil. Dirk Westhoff
Zweitgutachter: Prof. Dr. Thomas C. Schmidt

Eingereicht am: 28. August 2012

Tobias Markmann

Thema der Arbeit
Eine Android-Erweiterung zur Einschränkung von Rechteausweitung mittels dynamischer Darstellung von Informationsflüssen

Stichworte
Android, Rechteausweitung, Mobile Sicherheit, Informationsfluss, Interprozesskommunikation, TaintDroid

Kurzzusammenfassung
Apps für mobile Geräte haben eine steigende Verbreitung. Jedoch ist die Sicherheit sensibler Daten oft nur unzulänglich sichergestellt und es kann zu ungewollter Verbreitung dieser Daten kommen. Diese Arbeit befasst sich mit Rechteausweitung auf Android-Systemen, in Folge dessen sensible Daten missbraucht werden. Hierfür werden existierende Ansätze zur Einschränkung von Rechteausweitung durch Kommunikation verschiedener Apps analysiert und das eigene graphenbasierte Design zur Echtzeitdarstellung von Informationsflüssen zwischen Apps vorgestellt. Zur Überprüfung des Designs wurde eine Implementierung für die Android-Plattform auf Basis von TaintDroid umgesetzt und gegen Standardszenarien der Rechteausweitung evaluiert.

Title of the paper
An Android-Extension to Restrict Privilege Escalation Using a Dynamic Representation of Information Flows

Keywords
Android, privilege escalation, mobile security, information flow, inter-process communication, TaintDroid

Abstract
Apps on mobile devices getting more popular and widespread. However sensitive data is often insufficiently secured and becomes unintentionally distributed. This bachelor thesis is about misuse of sensitive data due to privilege escalation. For this existing approaches restricting privilege escalation due to communication between apps are analyzed and a new approach based on a graph-based design to model the real-time flow of information between apps is presented. To verify the design, it is implemented for the Android platform based on TaintDroid's modifications followed by an evaluation of the design based on common scenarios of privilege escalation.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1. Einleitung

1.1. Motivation

Das Android Betriebsystem erfreut sich seit Jahren steigender Beliebtheit, insbesondere auf Mobiltelefonen. Auch in anderen Bereichen, wie zum Beispiel Tablets oder SetTop-Boxen (Google TV) erhält es langsam Einzug.
Jeder kann sich für eine kleine Gebühr bei Google Inc. registrieren und danach Apps in Google Play[1] einstellen, ohne dass diese weder eine manuelle noch automatische Prüfung durchgehen. Dies stellt eine besondere Gefahr für den Nutzer dar, da sich einfach bösartige Apps in Google Play einstellen lassen, die vielen Millionen Nutzern sofort zur Verfügung stehen.
Beispiele hierfür sind Malware welche Telefonnummern ausspäht Cluley [2010] und ein Proof-of-Concept der BBC Ward [2010], welcher sich als Spiel tarnt um private Daten auszuspähen.
Nutzer sind es gewohnt, mit ein paar Klicks, einfach Anwendungen aus Google Play auf ihrem Gerät zu installieren und diese danach zu verwenden. Bevor Apps aus dem Google Play installiert werden, muss der Benutzer hingegen der Installation noch einmal zustimmen, wo ihm die erlaubten Rechte der zu installierenden App in einer Übersicht angezeigt werden.

Jedoch ist zu beachten, dass ein Großteil der Nutzer die Übersicht der Rechte vor der Installation nicht beachtet oder die Beschreibung der Rechte nicht versteht. In einer Studie [Felt et al., 2012, S. 11] war mehr als die Hälfte der Nutzer nicht bewusst, dass Ihnen die genauen Rechte, die eine Anwendung benötigt, aufgelistet wurden.
Aus diesem Grund müssen zusätzliche Maßnahmen getroffen werden, um Nutzer und deren Privatsphäre vor potentiellen Gefahren zu schützen.

[1] ehemals Android Market

Das Rechtesystem von Android stellt sicher, dass nur die bei der Installation von Anwendungen erlaubten Rechte bei der Ausführung einer App erlaubt werden. Jedoch beschränkt sich diese Überprüfung auf einzelne Anwendungen und nicht auf die Gesamtheit des Systems, d.h. alle zur Zeit laufenden Anwendungen.
Hierdurch können mehrere Anwendungen konspirierend ungehindert mehr Rechte erhalten als es dem Nutzer offensichtlich ist. Dieser Verbund kann mit Hilfe von offenen und geschlossene Kommunikationskanälen geschehen. Diese Kommunikation erlaubt es den Anwendungen Funktionalität, die durch deren jeweiligen Rechte erlaubt sind, an andere Teilanwendungen weiterzugeben. Die Konspiration von mehreren bösartigen Apps kann auch transitiv über mehrere Ebenen geschehen.
Ein Beispiel für eine App, welche aus zwei konspirierenden Teilen besteht ist Soundcomber [Schlegel et al., 2011]. Sie dient als Proof-of-Concept dafür, dass man mit zwei scheinbar harmlosen Anwendungen, eine nur für den Zugriff auf sensitive Daten (Telefongespräche), die andere nur mit Zugriff ins Internet, Kreditkartennummern von Smartphonenutzern ausspähen kann.
Gegen diese Angriffe auf die Sicherheit der Privatsphäre der Androidnutzer hat der Entwickler Google bisher noch keine Abwehrmaßnahmen unternommen.
Mit XManDroid [Bugiel et al., 2011] haben Wissenschaftler von der TU Darmstadt ein Überwachungssystem umgesetzt, welches ein Android-System zur Laufzeit überwacht und auf Basis von überwachten ICC-Kanälen versucht Rechteausweitungen zu bemerken und einzuschränken.

1.2. Android

Android ist ein Betriebsystem für Mobiltelefone und Tablets, das seit 2007 von Google und anderen Unternehmen entwickelt wird. Es basiert auf dem Linux-Kernel und übernimmt somit auch dessen, hauptsächlich von UNIX stammenden, Sicherheits- und Rechtemodelle.

Mit Dalvik gibt es auf Android eine, für mobile Einsatzzwecke optimierte, virtuelle Maschine für Programme. Diese werden meißtens in Java geschrieben. Obwohl Programme in Java geschrieben werden ist die Bytecode der Dalvik Virtuelle Maschine (DVM) inkompatibel zu dem der offiziellen Java VM. Aus diesem Grund bietet das Android Software Developer Kit (SDK) ein Programm an, um Java Bytecode in Dalvik Bytecode zu konvertieren.

Zur einfachen Kommunikation zwischen Apps untereinander und zu Systemdiensten dient das Binder Framework. Es besteht aus einer Kernel-Erweiterung und Schnittstellen im Android SDK und bietet App-Entwicklern RPC-Funktionalitäten ähnlich CORBA oder Java RMI. Ein Großteil der Schnittstellen im Android SDK basiert auf Binder. [Schreiber, 2011, S. 16]

1.3. Zielsetzung

Ziel dieser Arbeit ist es, auf Basis von Interprozesskommunikation entstandene, Rechteausweitung auf Android-Systemen zu erkennen. Die Arbeit konzentriert sich vor allem auf die Fälle von Rechteausweitung, die den ungewollten Zugriff auf sensitive Daten, wie Kontaktdaten oder Geoposition, ermöglichen.

Des Weiteren soll die hier entwickelte Android-Erweiterung die Entwicklung von Endnutzerapplikationen nicht beeinträchtigen. Durch die vollständige Implementierung auf System- und Application-Framework-Ebene müssen bestehende Apps nicht angepasst werden, um von den hier entwickelten Sicherheitsmechanismen zu profitieren. Somit lassen sich weiterhin Apps aus dem von der offiziellen Handelsplattform Google Play installieren und nur Apps die zur Laufzeit Rechteausweitung unternehmen könnten in ihrer Funktionalität beeinträchtigt werden.

Hierfür wird geprüft, ob auf Basis von der Analyse von Informationsflüssen, insbesondere das Weiterreichen von sensitiven Daten, mögliche Rechteausweitung zuverlässig erkannt werden. Das liefert die Grundlage für eine Unterbindung dieser Rechteausweitung.

1.4. Abgrenzung der Arbeit

Die Arbeit befasst sich Hauptsächlich mit Rechteausweitung auf Anwendungsebene und besonders dem Kommunikationssystem auf dieser Ebene. Diese kann absichtlich durch bösartige Apps entstehen und durch Programmfehler im Application-Framework oder der Apps.

Rechteausweitung auf Ebene des Systemkerns, meist ausnutzbar durch Programmfehler im Kernelcode, wird von dieser Arbeit nicht im Detail betrachtet. Es hat sich gezeigt, dass besehende Lösungen wie SELinux[2] oder auch TOMOYO Linux [Harada et al.,

[2] https://www.nsa.gov/research/selinux/index.shtml

2004] das Problem dieser Form der Rechteausweitung stark begrenzen können [Bugiel et al., 2011, S. 2]. Darüber hinaus wird davon ausgegangen, dass das Android-System, vor allem der Systemkern, Standarddienste und das Applikations Framework, inklusive der hier entwickelten Erweiterung, sicher und unverändert auf dem System laufen. Eine Modifizierung von diesen Komponenten gefährdet die zuverlässige Funktionsweise der Erweiterung.

Diese Absicherung kann zum Beispiel durch *Secure Boot* erfolgen. Es erlaubt einem ein Androidsystem vertrauenswürdig zu starten und auszuführen. Unter Nutzung eines Hardware-Kryptographiemoduls und kryptographischen Signaturen von festen Bestandteilen des Systems kann man somit sicherstellen, dass diese Bestandteile seit dem Zeitpunkt der Signierung nicht verändert wurden.
Durch die Signierung und konsequente Überprüfung von Bootloader, Kernel, Dalvik VM und weiteren sicherheitsrelevanten Teilen des Systems kann sichergestellt werden, dass die hier umgesetzten Modifikationen, welche der Erkennung von Rechteausweitung dienen, nicht ausgehebelt werden.

Da die Möglichkeiten, die eigentliche Rechteausweitung zu unterbinden oder anderweitige Gegenmaßnahmen einzuleiten, sehr vielseitig sind, stellen diese ein einzelnes Forschungsthema dar. Daher ist die Analyse und Konzeption möglicher Gegenmaßnahmen nicht Gegenstand dieser Arbeit.

1.5. Aufbau der Arbeit

In Kapitel 2 werden zuerst die grundlegenden Hintergrundinformation zum Thema Android, Prozesse auf dem System und die Kommunikation unter ihnen, erläutert. Ferner wird hier auch auf die Sicherheitsmechanismen von Android und Rechteausweitung eingegangen.

Kapitel 3 beschreibt bisherige Ansätze aus der Forschung, die sich zum Ziel gesetzt haben die Sicherheit auf der Android-Plattform zu verbessern und vor allem mögliche Rechteausweitungen einzuschränken. Darauf folgend wird das eigene Design auf Basis von dynamischer Darstellung von Informationsflüssen vorgestellt.

Kapitel 4 befasst sich mit den Implementationsdetails von TaintDroid und der Umsetzung der Android-Erweiterung, die das zuvor beschriebene Konzept der Darstellung von Informationsflüssen umsetzt. Ausserdem werden hier Hilfsprogramme vorgestellt, welche zur Visualisierung des jeweils aktuellen Systemzustands dienen.

In Kapitel 5 wird auf die Evaluierung der hier entwickelten Android-Erweiterung eingegangen. Es wird mit Hilfe von zwei Beispielszenarien überprüft ob die zuvor beschrieben Ziele erreicht wurden und sich die dynamische Darstellung von Informationsflüssen zur Einschränkung von Rechteausweitungen eignet.

Kapitel 6 liefert eine abschließende Übersicht über die Arbeit und die erreichten Ergebnisse.

Schließlich werden in Kapitel 7 mögliche Optionen für weitergehende Forschungsarbeiten offengelegt und vorgeschlagen.

2. Grundlagen

2.1. Android

2.1.1. Betriebsystem

Das Android Betriebsystem lässt sich in 4 Ebenen einteilen.

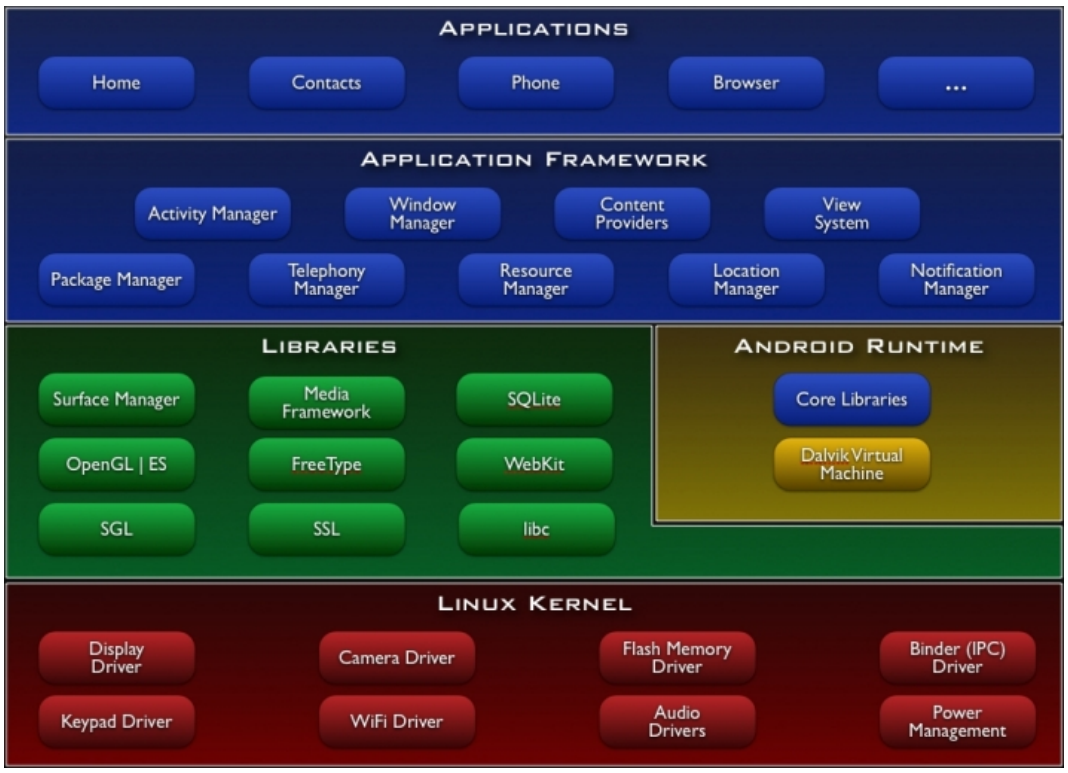


Abbildung 2.1.: Android System Architektur, (Quelle: [Google])

Auf unterster Ebene befindet sich der Betriebssystemkern, ein auf Linux basierender Kernel, der mit, für das Android System spezifischen Erweiterungen ergänzt wurde. Zu diesen Erweiterungen zählt unter anderem Binder, eine neue Schnittstelle zur In-

terprozesskommunikation (IPC), die flexible Kommunikation zwischen den laufenden Prozessen anbietet. Daneben gibt es noch Erweiterungen wie ashmem, für das einfachere Handhaben von geteiltem Speicher zwischen Prozessen.

Android ist keine Linux-Distribution im klassischen Sinne. Es unterstützt weder das sonst gängige X-Window-System als graphische Benutzeroberfläche, noch sind standardmäßig die üblichen GNU Bibliotheken vorhanden.

Native Bibliotheken wie z.B. Datenbanken, Web-Rendering oder auch OpenGL ordnen sich in der mittleren Ebene ein. Die Middleware von Binder befindet sich auch auf dieser Ebene. Binder ist mit bekannten RPC Umsetzungen wie CORBA oder auch Java RMI vergleichbar. Der wesentliche Unterschied ist, das Binder nicht Netzwerktransparent ist. Es wurde nur für Interprozesskommunikation innerhalb eines Systems und nicht über Netzwerkgrenzen hinaus entwickelt.

Ausserdem ist hier auch Dalvik einzuordnen. Dalvik ist eine register-basierte VM, die sich an der ARM-Architektur orientiert. Sie ist optimiert für ressourcenschwache Geräte, da Android hauptsächlich für den Embedded-Bereich wie Smartphones oder auch Tablets ausgelegt ist. [Nicola Carlo, 2009]

Auf der oberen Ebene ist der Großteil des Android Applikation Frameworks angesiedelt, gegen das die Entwickler von Benutzeranwendungen programmiert. Diese Apps befinden sich auf der obersten Ebene.

2.1.2. Aufbau von Android Apps

Der Aufbau von Apps, d.h. Endnutzeranwendungen, auf der Android-Plattform unterscheidet sich grundlegend von dem, auf üblichen Desktop-Systemen. Bei gängigen Unix-Anwendung hat man nur einen Einstiegspunkt in die Anwendung, üblicherweise die *main()*-Funktion. Eine Android App hingegen kann aus mehreren high-level Komponenten bestehen, von denen jede einem möglich Einstiegspunkt in die App entspricht. [Mednieks et al., 2011, S. 75f.]

Der Aufbau jeder App wird in ihrer Manifest-Datei beschrieben. Die Manifest-Datei hat ein XML-basiertes Format, wodrin die einzelnen Komponenten aufgeführt sind und welche Aktionen mit der Anwendung an welche Komponente weitergeleitet werden. Ausserdem beschreibt die Manifest-Datei die benötigten Berechtigungen der App.

Die möglichen Komponenten einer App werden im Folgenden genauer erläutert.

2.1.2.1. Activity

Activity-Komponenten sind die zentralsten Komponenten einer App. Nur über sie findet eine direkte Interaktion mit dem Benutzer statt. Aussehen und Funktionalität einer Activity ist strikt getrennt, indem ersteres deklarativ in einer XML-Datei beschrieben wird und letzteres durch eine Java-Klasse, die von der *Activity*-Klasse des Application Frameworks erbt, umgesetzt wird. Der Einstiegspunkt in eine Activity-Komponente ist dessen *onCreate()*-Methode. [Mednieks et al., 2011, S. 77f.]

Activities können durch Intent-Nachrichten, ein high-level IPC-Mechanismus, ausgeführt werden und Werte an den aufrufenden Code zurückliefern. Das ermöglicht eine hohe Wiederverwendbarkeit der Komponenten und erleichtert dadurch das Erstellen neuer Apps. Ein Beispiel hierfür sind Activities zum Anzeigen einer geographischen Position auf einer Karte oder von Kontaktdaten aus dem Adressbuch.

2.1.2.2. Service

Service-Komponenten dienen hauptsächlich zur Ausführung von Hintergrundaufgaben, wie das Herunterladen von Dateien oder das Abspielen von Musik. Services können somit längere Laufzeiten haben als Activities, jedoch besitzen sie keine Benutzeroberfläche. Wenn der verfügbare Speicher knapp wird kann das System jedoch Services von aussen beenden um Speicher für andere Komponenten des Systems zur Verfügung zu stellen.

Methoden von Services werden via Binder-RPC und Android Interface Definition Language (AIDL) anderen Komponenten der eigenen App oder auch der anderer Apps angeboten. [Mednieks et al., 2011, S. 79]

2.1.2.3. ContentProvider

ContentProvider-Komponenten dienen als Speicher für persistente Daten. Sie sind systemweit, unter einer im Manifest beschriebenen Uniform Resource Identifizier (URI), Adressierbar und können auf diese Weise auch zur Kommunikation zwischen verschiedenen Anwendungen verwendet werden. Es gibt eine feste Menge von Operationen, die auf einen ContentProvider ausgeführt werden können: Insert, Quert, Update und Delete.

Beispiele für ContentProvider, welche mit Android standardmäßig mitgeliefert werden, sind unter anderem die ContentProvider für Kalender, Anruflisten oder auch Musik. [Mednieks et al., 2011, S. 79f.]

2.1.2.4. BroadcastReceiver

BroadcastReceiver-Komponenten fungieren als Empfänger von high-level IPC Nachrichten, den Intents. Gesendete Intents können von mehr als einem BroadcastReceiver empfangen werden. Anwendungsentwickler können für jeden BroadcastReceiver Filter in der Manifest-Datei hinterlegen, sodass der BroadcastReceiver nur für vorher bestimmte Ereignisse aufgerufen wird.

Ein Beispiel für Ereignisse, die mittels Broadcast-Intent verteilt werden, ist der Empfang einer SMS. [Mednieks et al., 2011, S. 82]

2.1.3. Rechtemodell

Auf einem Android-System hat man mit verschiedenen Rechtemodellen auf den verschiedenen Ebenen des Systems zu tun. Auf der untersten Ebene gibt es, wie unter Linux üblich, auf UIDs und GIDs basierende Datei- und Ausführungsrechte. Diese werden durch den Betriebssystemkern durchgesetzt. Die Endbenutzer-Anwendungen laufen alle als dedizierten Prozesse. Bei der Installation einer neuen App wird Ihr vom System eine feste UID zugewiesen unter der diese im System ausgeführt wird. Die UID ist Entwicklerspezifisch, das heisst alle Anwendung des gleichen Entwicklers (durch den selben Entwicklerschlüssel signiert), laufen unter der selben UID.

Darüber hinaus gibt es auf weiter oberen Ebenen Android spezifische Berechtigungen (*Permissions*). In Android 2.2 gibt es 134 Permission , welche sich in 3 Kategorien [Felt et al., 2011a, S. 2] einteilen lassen:

1. Normal permissions, für Funktionen, von denen keine direkte Gefahr ausgeht.
2. Dangerous permissions, welche für den Zugriff auch private Daten oder Funktionen, die mit Gebühren verbunden sind, benötigt werden.
3. Signature/System permissons, benötigt unter anderem zum entfernen von installierten Apps.

Ausserdem können Entwickler auch anwendungspezifsche Permissions erstellen, welche die Kommunikation mit ihren Komponenten absichern kann.

Für Endnutzeranwendungen stehen dem Nutzer von Android unterschiedliche Quellen zur Verfügung. Die meistgenutzte Quelle ist die standardmäßig installierte App *Google Play*. Anders als beim Apple AppStore, wo die von Entwicklern eingereichten Anwendungen vom Betreiber der Verkaufsplattform verifiziert und getestet werden, werden bei Google Play die eingereichten Apps direkt den Endnutzer bereitgestellt.

Um mögliche Gefahren für die Privatsphäre und Sicherheit der Nutzer von Android zu Beschränken, muss der Nutzer vor der endgültigen Installation einer App diesen Installationswunsch in einem Dialog explizit bestätigen. In diesem Dialog werden dem Nutzer die Berechtigungen, inklusive kurzer Erläuterungen zu denen, welche die App benötigt aufgezeigt. Hier hat der Nutzer die letzte Möglichkeit über die Installation zu entscheiden und muss zwischen Funktionalität und Sicherheit abwägen. Damit hat Google die Frage der Sicherheit und Privatsphäre des Nutzers an den Nutzer selbst weitergeleitet, welche jedoch meist die einzelnen Berechtigungen nicht verstehen oder den Dialog ganz ignorieren. [Felt et al., 2012]

Ein Großteil der Permissions wird innerhalb des Systemprozesses von Android implementiert, wo sich die Implementation der meisten Dienste befindet. Nur ein paar Permissions, welche sich einfach zu Funktionen des Systemkerns zuweisen lassen, werden mit Hilfe von Unix-Gruppen und Zugriffskontrolle dieser auf Systemkernebene erzwungen. Zu diesen Ausnahmen zählen unter anderem Bluetooth- und Internetzugriff. [Felt et al., 2011a, S. 3]

2.1.4. Kommunikationswege

Auf einem Android-System verhalten sich die Apps und System-Dienste wie in einem verteilten System. Aus Sicherheitsgründen läuft jede Anwendung in ihrer eigenen Instanz der DVM, welche als limitier Prozess direkt auf dem OS läuft. Da diese Prozesse nur Zugriff auf die eigenen Daten haben, stellen sie eine Sandbox dar. Um dennoch einen hohen Grad an Wiederverwendbarkeit der Komponenten zu ermöglichen, ist man auf Kommunikation zwischen den Diensten und Apps angewiesen. Abbildung 2.2 gibt eine Übersicht über die Möglichkeiten der Kommunikation die einer Android-App zur Verfügung stehen.

Der offizielle Weg unter Android eine Kommunikation zwischen Anwendungen herzustellen sind Intents oder RPC-Aufrufe mittels Binder.

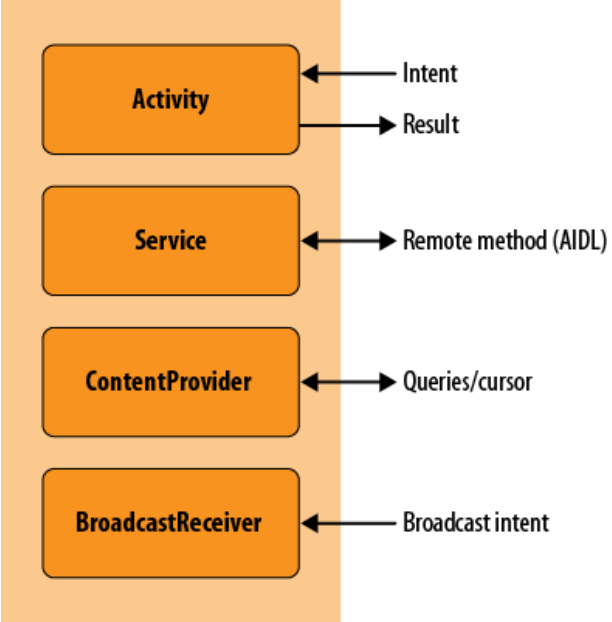


Abbildung 2.2.: Kommunikationsmöglichkeiten von Android-Komponenten (Quelle: [Mednieks et al., 2011, S. 83])

2.1.4.1. Binder

Mit Binder gibt es auf Android eine, vom Kernel unterstützte, Art der Interprozesskommunikation die im üblichen Linux-Kernel nicht vorhanden ist.

Auf dieser Basis bietet Android den Systemdiensten und Endanwendungen zwei verschiedene Modelle zur Kommunikation zwischen Komponenten der gleichen oder verschiedenen Anwendung an: **RPC** und **Intents**.

2.1.4.2. RPC

Eine der Hauptfunktionalitäten von Binder ist es, Objekte und deren Methoden aus einem Prozess (z.B. einer DVM Sandbox) einem anderen Prozess zur Verfügung zu stellen. Im lokalen Prozess können Methodenaufrufe auf entfernte Objekte von denen auf lokale Objekte abstrakt nicht unterschieden werden, sodass Apps oder andere Komponenten des Systems nicht wissen müssen ob ein Service im lokalen Prozess oder von einem dedizierten Prozess implementiert ist. [Schreiber, 2011, S. 10-14]
Des Weiteren bietet Binder eine eigene IDL namens AIDL an, mit der die Schnittstellen der Objekte, die zur Verfügung gestellt werden sollen, in einer am Java-Syntax ange-

lehnten Sprache definiert werden. Auf Basis dieser Beschreibung generiert das Android SDK dann die passenden Proxy- und Stub-Klassen. [Schreiber, 2011, S. 17-18]

2.1.4.3. Intents

Intents sind stark abstrahierte Nachrichten die Entwicklern vorrangig als Java-API zur Verfügung stehen und erlauben eine sehr hochrangige Art der Kommunikation zwischen Komponenten des Systems. Speziell werden sie hauptsächlich zum Starten von neuen Activities oder zur Kommunikation zwischen Activities verwendet. Ein Intent beinhaltet eine Aktion die auszuführen ist, z.B. einen Telefonanruf auszuführen, und optional zusätzliche Daten, z.B. welche Nummer anzurufen ist.
Ein Intent ist entweder explizit, d.h. an eine vordefinierte Komponente gerichtet, oder implizit. Bei impliziten Intents bestimmt das System an welche Komponenten es ausgeliefert wird. [Schreiber, 2011, S. 16]

Eine besondere Form von Intents auf der Android-Plattform sind die Broadcast Intents. Diese können von mehr als eine App empfangen werden und werden oft zu Benachrichtigung von Apps und Systemkomponenten über besondere Ereignisse im System verwendet. Installation von neuen Apps aber auch ein niedriger Batteriestand zählen zu derartigen Ereignissen.

2.1.4.4. Queries/Cursor

ContentProvider stellen eine wichtige Komponente innerhalb von Android dar und sind essentiell zur Umsetzung eines Model-View-Controller (MVC) Stils unter Android [Mednieks et al., 2011, S. 81]. Abfragen werden mittels *query()*-Methode ausgeführt und auf die jeweiligen Ergebnismengen kann mit Hilfe eines Cursor-Objekts iteriert werden. Die von ContentProvidern angebotenen Aktionen umfassen das Erstellen neuer, das Entfernen oder Ändern bestehender Einträge und die flexible Abfrage von Einträgen [Mednieks et al., 2011, S. 306f.].

Der eigentliche Zugriff auf die Aktionen der ContentProvider wird über die im vorherigen Abschnitt 2.1.4.2 beschriebe RPC-Funktionalität geregelt.

2.2. Rechteausweitung

Unter Rechteausweitung versteht man die Erweiterung der Rechte einer Komponente über die, vorher für dieser Komponente, definierten Rechte hinaus. Es gibt verschiedene Möglichkeiten für Komponenten ihre Rechte zu erweitern.

Rechteausweitung lässt sich in zwei Arten aufteilen:

Vertikale Rechteausweitung Hierbei kann eine Anwendung oder Benutzer durch Bugs oder Designfehler Funktionen höherer Ebene ausführen, für die normalerweise mehr Rechte von Nöten wären.

Horizontale Rechteausweitung Diese Art von Rechteausweitung tritt ein, wenn Anwendungen oder Benutzer auf gleicher Ebene auf Funktionen und Daten anderer Anwendungen oder Nutzer auf dieser Ebene zuzugreifen können.

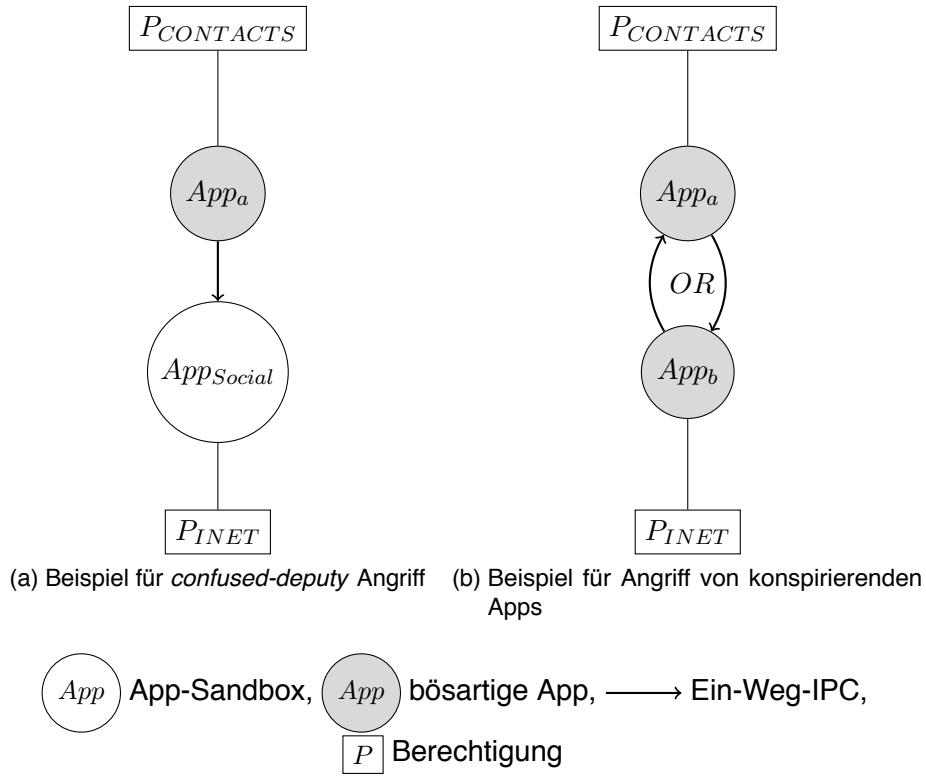
Ein Prozess kann Fehler im Betriebssystemkern ausnutzen um beliebige Instruktionen mit Systemrechten auszuführen und damit die Rechte des eigenen Prozesses verändern. Derartige Angriffe finden vertikal, über mehrere Systemeben hinweg, statt und können durch SELinux eingeschränkt werden.

Die Apps auf einem Andorid-System laufen in ihren eigenen Sandboxen, welche auf Systemkern-Ebene durch individuelle UIDs von anderen Apps und dem System selbst getrennt sind. Somit ist das System vor bösartigen oder fehlerhaften Anwendungen geschützt. Ausserdem sind die Apps gegen andere Apps geschützt, da jede App nur Direktzugriff auf die eigenen Daten innerhalb ihrer Sandbox hat.
Um in einem System, wo jede App für sich gekapselt ist, dennoch Kollaboration mit anderen Komponenten zu ermöglichen gibt es Kommunikationsmöglichkeiten zwischen den Apps. Ohne diese wären die Möglichkeiten der Entwicklung von brauchbarer Software auf dieser Art von Systemen sehr beschränkt.

Allerdings sind es genau diese Kommunikationsmöglichkeiten zwischen den Apps, die eine Schwäche für die Sicherheit des Systems und damit auch der Privatsphäre des Nutzers darstellen. Die Schwäche liegt darin, dass die Kommunikation zwischen verschiedenen Komponenten unkontrolliert verläuft und somit eine horizontale Rechteausweitung begünstigen kann.

Dabei unterscheidet man zwischen *confused-deputy* Angriffen und Angriffen von konspirierenden Komponenten.
Confused-deputy Angriffe werden gerade dann ermöglicht, wenn die Entwickler von

Komponenten die Herkunft von Anfragen dritter nicht auf Berechtigung prüfen und auf diese Weise einen Missbrauch zulassen. Bei Angriffen konspirierenden Komponenten (engl. *colluding applications*) erlangen mehrere eingeschränkte Komponenten, welche einzeln keine gefährliche Menge an Berechtigungen hält, gemeinsam durch uneingeschränkte Kommunikation untereinender gemeinsam eine größere, potentiell gefährliche, Menge an Rechten.



(a) Beispiel für *confused-deputy* Angriff (b) Beispiel für Angriff von konspirierenden Apps

Abbildung 2.3.: Rechteausweitungsangriffe zwischen Anwendungen

Abbildung 2.3a zeigt ein Beispiel für einen *confused-deputy* Angriff. Die Schnittstelle von App_{Social}, auf die die bösartige Anwendung App_a Zugriff hat, führt unzureichende Zugriffskontrollen durch, wodurch App_a private Kontaktdaten über das Internet versenden kann.
In Abbildung 2.3b hingegen sind die zwei bösartigen Anwendungen App_a und App_b zu sehen, welche, so bald auch nur in eine Richtung zwischen den Anwendung kommuniziert werden kann, gemeinsam eine riskante Kombination an Rechten erhalten.

Android selbst hat keine ausreichenden Gegenmaßnahmen bezüglich Rechteausweitungsangriffen in Form von *confused-deputy*-Angriffen oder konspirierenden Apps. Es stellt lediglich den direkten Zugriff auf die diversen API-Schnittstellen, innerhalb der Implementation, in Anbetracht der jeweiligen Rechte einer Anwendung, sicher.

Indirekter Zugriff auf die API des Android SDK, durch Kommunikation zwischen den Apps, ist auf einem unveränderten Android-System möglich.

2.3. Angreifermodell

Einem potentiellen Angreifer stehen verschiedene Möglichkeiten zur Verfügung um seinem angreifenden Prozess(-en) mehr Rechte zu Nutzen als ihm ursprünglich zugesichert waren.
Ein Angriff besteht in diesem Fall aus einer App die Zugriff auf private Daten, wie z.B. die aktuelle Position oder SMS-Nachrichtenspeicher, und einer App welche Rechte besitzt mit der Außenwelt zu kommunizieren, z.B. Internetzugriff oder Senden von SMS. Es ist nicht allgemein erforderlich, dass beide Anwendungen bösartig sind. Es reicht auch aus, wenn es eine Schwachstelle in einer bestehenden Anwendung mit entsprechenden Rechten gibt, welche sich durch einen Angreifer ausnutzen lässt.

Unabhängig davon, ob zwei Anwendungen konspirieren oder eine bösartige App eine Schwachstelle einer weiteren Anwendung ausnutzt, kommt eine Rechteausweitung erst durch eine Kommunikation zwischen beiden Anwendungen zustande.

Mit Soundcomber haben Schlegel et al. demonstriert, dass die Bedrohung, die von Trojanern die auf die Android-Plattform ausgerichtet sind, ernst zu nehmen ist. Mittels zwei unscheinbaren Anwendungen, einer Sprachwahl-App oder Diktier-App und einer beliebigen Anwendung mit Berechtigung zum Internetzugriff. [Schlegel et al., 2011, S. 2] gelingt es Ihnen Kreditkartennummern, die während Telefonbankings verwendet werden, abzufangen und ins Internet zu verschicken. Damit die App mit Internetzugriff die gesammelten Daten an den Autor des Trojaners zurücksenden kann, bedarf es einer Kommunikation zwischen den beiden Apps. [Schlegel et al., 2011, S. 10]

Außerdem stellen Schlegel et al. verschiedenste versteckte Kommunikationskanäle vor, mit der die gesammelten Daten übertragen werden könnten. Die erwähnten Kanäle haben alle nur eine geringe Bandbreite, was jedoch bei Soundcomber kein Problem ist, da die Erkennung der Kreditkartennummern aus dem aufgenommenen Ton schon in der aufnehmenden App passiert.

3. Design

3.1. Bisherige Ansätze

Zur Abschwächung von Rechteausweitungsangriffen auf Android-Systemen gibt es bereits verschiedene Ansätze. Der umfassendste Ansatz, welcher sich hauptsächlich gegen Rechteausweitungangriffe auf Anwendungsebene richtet, stammt von Bugiel et al. namens XManDroid.

3.1.1. XManDroid

XManDroid [Bugiel et al., 2011] ist eine Erweiterung des Android Frameworks und der darin enthaltenen Middleware zur IPC.

Abbildung 3.1 zeigt die Architektur der XManDroid-Erweiterung. Als zentrale Komponente gibt es den *SystemView*, welche einen Graphen pflegt, der den aktuelle Systemzustand darstellt. Darin enthalten sind die zurzeit installierten Komponenten, dargestellt als Knoten, und die, in der Vergangenheit erlaubten Kommunikationen unter ihnen, dargestellt als ungewichtete ungerichtete Kanten. Die Komponenten unterscheiden sich in Sandboxen für Endnutzer-Anwendungen und virtuelle Komponenten für relevante Teile des Systems.[Bugiel et al., 2011, S. 6f.]

Zudem wurde ein Regelwerk eingeführt. Mit diesem können komplexe Regeln formuliert werden, die beschreiben, unter welchen Bedingungen Zugriffe auf bestimmte oder allgemein Komponenten erlaubt sind. Die Verbesserung der Sicherheit auf dem System, die durch die gesamte Erweiterung erreicht wird, ist stark von den spezifizierten Regeln abhängig.[Bugiel et al., 2011, S. 5f.]

Um die Durchsetzung der Regeln zu ermöglichen haben Bugiel et al. Androids Referenzmonitor innerhalb des ActivityManagerService angepasst. Nur im Fall einer positiven Entscheidung des ursprünglichen Referenzmonitors, wird eine Entscheidung von XManDroid angefragt. Diese bestimmt dann, auf Basis von bisherigen gecachten Entscheidungen, dem aktuellen Graphen, der Berechtigungen der Anwendungen

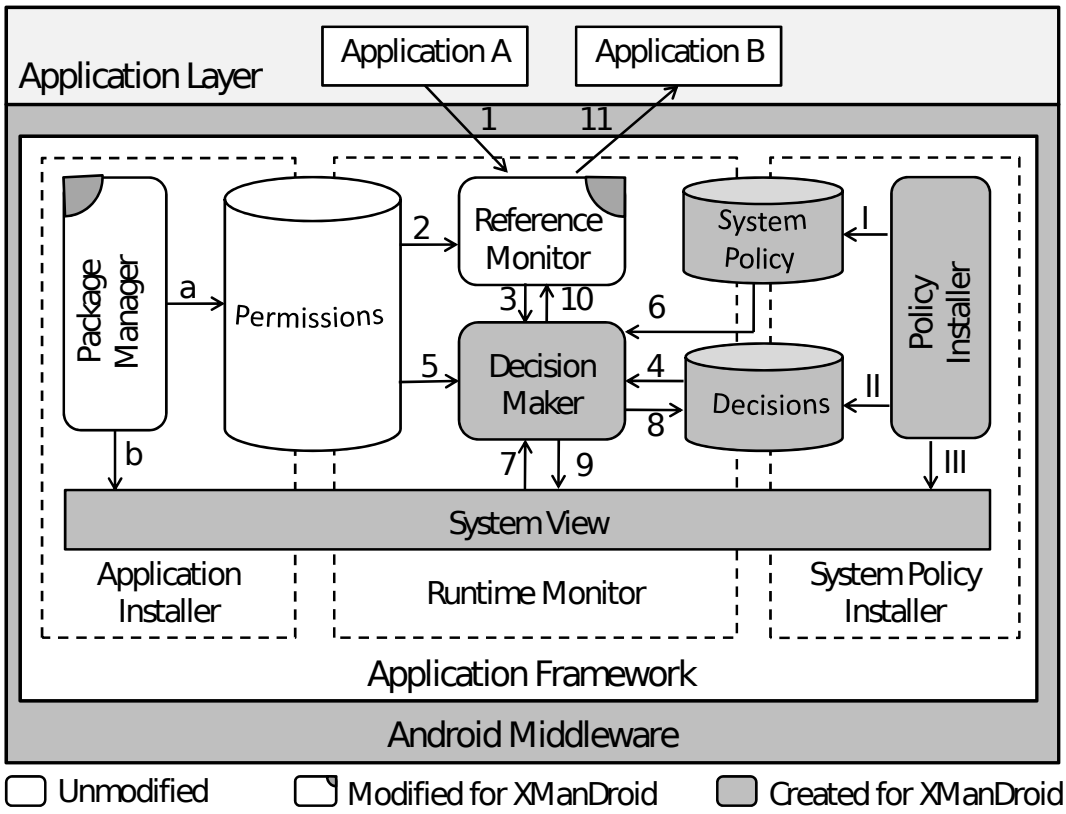


Abbildung 3.1.: XManDroid Architektur (Quelle: [Bugiel et al., 2011, S. 5])

und des Regelwerks, ob der Zugriff erlaubt ist oder nicht. Dies ist dann die endgültige Entscheidung des Referenzmonitors. [Bugiel et al., 2011, S. 5]

Laut Bugiel et al. haben Tests gezeigt, dass es besonders wichtig ist, vor allem bei Content Providern und Service Providern, zwischen Lese- und Schreibzugriffen zu unterscheiden, um den Anteil der False-Positives zu beschränken. [Bugiel et al., 2011, S. 13]

Erweiterung

Bugiel et al. haben später eine Erweiterung für XManDroid vorgeschlagen, welche ihr Verfahren um Zugriffskontrolle über das Dateisystem, Unix Domain Sockels und IP-Sockets erweitert [Bugiel et al., 2012].

Der Graph wurde zu diesem Zweck um Dateien/Unix Domain Sockels und Internet-Sockets in Form von Knoten erweitert. Ein weiterer großer Unterschied zum vorherigen

Ansatz von XManDroid ist es, dass es sich nun um einen gerichteten Graphen handelt. Kanten für Interkomponentenkommunikation sind mit dieser Erweiterung bidirektional und für Zugriffe auf Dateien und Sockels wurden gerichtete Kanten hinzugefügt um zwischen Lese- und Schreibzugriffen differenzieren zu können. [Bugiel et al., 2012, S. 7]
Um die Zugriffe genau überwachen zu können, wurde eine kernseitige MAC, in Form von *TOMOYO Linux*, in den Android-Kern eingebaut. Die Schnittstellen von *TOMOYO Linux* wurden dahingehend angepasst, dass es Zugriffe auf Dateien und Sockels in Systemkernebene an XManDroid weiterreicht und im Gegenzug Änderungen, für das eigene kernseitige Regelwerk, entgegennimmt. [Bugiel et al., 2012, S. 10]

Zusätzlich werden auch Intent-Nachrichten zwischen Komponenten auf einem Android-System derartig markiert, sodass dem Referenzmonitor die Aufrufskette, die UIDs der involvierten Apps, zur Verfügung steht, die zu der Intent-Nachricht geführt hat. Diese Aufrufsketten werden in den *SystemView* integriert und kann zusätzlich zur Entscheidungsfindung innerhalb XManDroids herangezogen werden. Dabei wurden Bugiel et al. durch *Quire* [Dietz et al., 2011] angeregt. [Bugiel et al., 2012, S. 6]

3.1.2. IPC Inspection

Felt et al. haben mit IPC Inspection ein Design umgesetzt, was darauf abzielt wiederholte Delegation von Berechtigungen durch IPC einzuschränken. Ihr Ansatz ist unabhängig von der Laufzeitumgebung und wurde von ihnen auf Android und in einer Browserumgebung umgesetzt. Apps haben nur die Berechtigungen, die sie explizit vom Benutzer zugewiesen bekommen haben. [Felt et al., 2011b, S. 2]

Beim Empfang von IPC Nachrichten einer App werden die Berechtigungen der empfangenden App angepasst. Dabei wird sichergestellt, dass die empfangene App die Nachricht nur mit Berechtigungen verarbeiten kann, die nicht über die der initiierenden App hinausgehen. Bei dieser potentiellen Einschränkung von Berechtigungen, entsprechen diese der Schnittmenge aus den Rechten der Apps in der Aufrufskette. Somit können Berechtigungen nur verringert und nicht erweitert werden. [Felt et al., 2011b, S. 7]

Da sich diese Arbeit nur auf Einschränkung von Rechteausweitung auf der Android-Plattform befasst, wird im Folgendem nur auf die Android-Umsetzung von IPC Inspection eingegangen.

Im Falle von einer Interaktion zwischen Komponenten verschiedener Apps, wird dies vom *ActivityManager* an den *PackageManager* weitergereicht, welcher überprüft ob

die empfangende App in ihren Berechtigungen beschränkt werden muss. Wenn die empfangende App mindestes alle Berechtigungen des Initiators hat oder eine Systemkomponente ist , wird die Nachricht, wie auf einem Android-System üblich, einfach weitergereicht. Im dem Fall, dass die Berechtigungen reduziert werden müssen, wird vom *ActivityManager* eine neue Instanz gestartet, an dem die Nachricht weitergeleitet wird. Ist es unmöglich eine neue Instanz zu starten, zum Beispiel bei Service-Komponenten von denen es nur eine Instanz geben darf, werden zur Laufzeit die Berechtigungen der laufenden App reduziert. [Felt et al., 2011b, S. 9]

Dadurch das neue Instanzen von den Apps gestartet werden, sind die bestehenden Instanzen von den Änderungen ihrer Berechtigungen nicht berührt. Das garantiert, dass die Funktionalität der Apps soweit wie möglich erhalten bleibt. [Felt et al., 2011b, S. 8]

3.1.3. TaintDroid

Um den Fluss sensitiver Daten auf einem Android-System zu verfolgen und das Versenden dieser Daten an Dritte aufzuspüren, haben Enck et al. TaintDroid entwickelt. [Enck et al., 2010]

Dafür wurde Android in zwei Bereichen angepasst. Zum einen wurde die Dalvik VM um Möglichkeiten zur dynamischen Verfolgung von Markierungen erweitert, indem die Markierungen über Variablen und Methoden hinweg propagiert werden. Die Markierung einer Variable oder Nachricht, auch TaintTag genannt, hat in TaintDroid eine Größe von 32 Bit und ermöglicht somit das gleichzeitige Markieren mit 32 verschiedenen Markierungstypen.
Ausserdem werden die Markierungen über Dateien und Nachrichten, die zwischen den Apps selbst und auch dem System ausgetauscht werden weitergereicht. [Enck et al., 2010, S. 3]

Gerade diese Funktionalität ermöglicht, durch darauf aufbauende Tools, Rechteausweitung zu erkennen.
Zum anderen wurden Teile des Android-Frameworks angepasst, um Quellen und Senken für die Markierungen zu definieren. Die möglichen Quellen müssen manuell für das Android-System spezifiziert und deren Quellcode angepasst werden, um die Rückgabewerte von deren Schnittstellen mit dem passenden TaintTag zu versehen.

Die Autoren von TaintDroid klassifizieren Markierungsquellen in folgende Kategorien [Enck et al., 2010, S. 9f.]:

1. Sensoren mit geringer Bandbreite (Beispiel: Geoposition, Beschleunigungssensor)

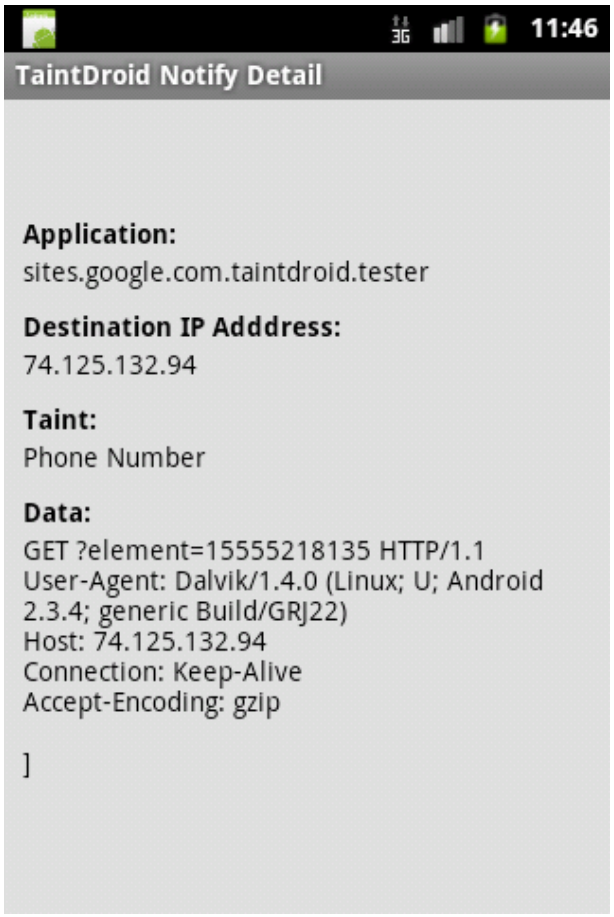


Abbildung 3.2.: Darstellung von sensitiven Daten, die das System verlassen im TaintDroid Notifier

2. Sensoren mit hoher Bandbreite (Beispiel: Mikrofon, Kamera)
3. Datenbanken (Beispiel: Content Provider für SMS oder Adressbuch)
4. Gerätekennung (Beispiel: IMSI, IMEI)

Die einzige Senke, die von Enck et al. implementiert wurde, ist das Netzwerkinterface. Hierfür wurden die Socket-Klassen innerhalb des Android SDKs angepasst. [Enck et al., 2010, S. 10]

TaintDroid selbst sieht es nicht vor Maßnahmen gegen das Weiterleiten sensitiver Daten an Dritte zu Unternehmen. Beim passieren von markierten Daten in einer, von den Autoren angepassten, Senke wird diese Tatsache auf der Systemkonsole protokolliert.

In Abbildung 3.2 sieht man mit TaintDroid Notifier, eine Beispielanwendung der Entwickler, welche die Systemkonsole laufend auf TaintLog-Ereignisse hin überwacht und bei Verlassen von sensitiven Daten über das Netzwerkinterface den Nutzer informiert.

3.1.4. Zusammenfassung

Sowohl XManDroid, als auch IPC Inspection, beschränken sich Hauptsächlich auf Einschränkung der Rechte auf Basis der Permissions der kommunizierenden Apps ab.

Jedoch weißt das Berechtigungssystem auf Applikationsebene von Android einige Schwächen auf. Laut Felt et al. sind viele Berechtigungsanforderungen der Android API falsch oder gar nicht dokumentiert. Unklarheiten und Inkonsistent in der Strukturierung und Granualität der Permissions führt zu Konfusion bei Anwendungsentwicklern. [Felt et al., 2011a, S. 5f.]

Der Ansatz von TaintDroid hingegen basiert lediglich auf dynamische Propagierung von Variablenmarkierungen zur Laufzeit im System. Auf diese Weise lässt sich der Verlauf von, durch Berechtigungen geschützten, sensitiven Daten durch das System in Echtzeit verfolgen. Aus diesem Grund wurde der hier entwickelte Ansatz auf Basis von TaintDroid umgesetzt.

3.2. Zuverlässigkeit der Erkennung von Rechteausweitung

Die Zuverlässigkeit eines Systems zur Erkennung von Rechteausweitung lässt sich Anhand dessen Entscheidungen über bestimmtes Verhalten von Apps bestimmten. Durch die Entscheidung des Systems und das Verhalten der App ergibt sich folgende Matrix:

Die Anteile in der Wahrheitsmatrix [spr, 1998], die in Tabelle 3.1 zu sehen ist, sind stark Abhängig von der Definition von Rechteausweitung für ein System und dessen Abgrenzung zu anderen Systemen.

Für Ansätze die explizit zur Einschränkung von Rechteausweitung entwickelt wurden sind üblicherweise hohe Werte für *richtig positiv* und *richtig negativ* Erkennung zu erwarten, da dies ihr Schwerpunkt ist. Besonders hervorzuheben sind hier die zwei übrigen Fälle:

	App führt erweiterte Rechte aus	App nutzt nur die eigenen Rechte
System erkennt Rechteausweitung	richtig positiv	falsch positiv
System erkennt keine Rechteausweitung	falsch negativ	richtig negativ

Tabelle 3.1.: Wahrheitsmatrix der Systemzuverlässigkeit

falsch positiv Ist dieser Anteil für ein System zu hoch ist mit sehr eingeschränkter Benutzbarkeit zu rechnen. Das Verhalten vieler legitimer Apps wird fälschlicherweise als illegale Rechteausweitung erkannt

falsch negativ Dieser Anteil steigt mit der Zeit an, wenn neue Angriffe, die in die Definition der Rechteausweitung für das System fallen, entdeckt werden, die noch nicht vom System als solche erkannt werden.

Im Vergleich zu XManDroid, ist ein geringerer Anteil von False-Positives zu erwarten, da bei der hier entwickelten Erweiterung auch die Veränderung von verschiedenen Datenspeichern im System modellieren und nicht nur die verschiedenen offenen und geschlossenen Kommunikationskanäle. Somit können die erlaubten und verbotenen Anwendungsfälle bzw. Anwendungsmuster exakter spezifiziert werden.
Stark ausschlaggebend für die Anzahl der False-Positives ist die vom System verwendete Policy-Datei, dessen Regeln bestimmen welche Verbindungen zwischen welchen Anwendungen erlaubt sich und welche nicht.

Bestimmte Systemteile und Anwendungen stellen eine besondere Quelle von False-Positives dar. Hierzu gehören vor allem System Settings Content Provider und Application Launcher. Die Erkennung kann jedoch durch Analyse des Inhalts der ICC[1] Nachrichten und dessen Filterung weit reduziert werden. [Bugiel et al., 2011, S. 13]

3.3. Zielsetzung

Es soll eine Erweiterung für die Android-Plattform geschaffen werden, welche auf Basis von dynamisch Informationsflüssen zur Laufzeit eine Systemrepräsentation zur Verfü-

[1] Inter-Component Communication

gung stellt. Ziel ist es dann auf Basis dieser Darstellung des aktuellen Systemzustands Fälle von ungewollter Rechteausweitung zu erkennen und zu unterbinden, welche eine Gefahr für die Privatsphäre des Endnutzers darstellt.

Hierzu wird auf TaintDroid [Enck et al., 2010] zurückgegriffen, welches sensitive Daten an den Quellen markiert, die Markierungen durch die DVM ausbreitet und bei Kommunikation mit anderen Systemkomponenten, diese Pakete auch markiert. Androids Middleware, welche im Zentrum der Kommunikation zwischen Komponenten steht, wird derart Angepasst, der Systemzustand laufend mit der aktuellen Kommunikation übereinstimmt.

Im Vergleich zu XManDroid und IPC Inspection wird in diesem Ansatz die Kommunikation zwischen den Komponenten der Apps noch feingranularer erfasst und ins System übernommen. Dies wäre ohne die Markierungen von TaintDroid nicht möglich, womit an den Kommunikationsschnittstellen geprüft werden kann, ob Daten einer sensitiven Quelle entstammen.

3.4. Systemrepräsentation

Angelehnt an XManDroid wird das System auch als Graph dargestellt. Jedoch ist hier der Systemgraph wesentlich dynamischer und ändert sich laufend mit der Benutzung des Systems.

3.4.1. Dynamischer Systemgraph

In Abbildung 3.3 sieht man ein abstrahiertes Beispiel für eine Momentaufnahme der Systemdarstellung. Zur Zeit laufen die drei Apps $App_{AddrBook}$, $App_{Recorder}$ und App_{Widget}. Die jeweiligen Permessions der einzelnen Apps sind als P_α dargestellt und mit der jeweils dazugehörigen App mit verbunden.

Der Systemgraph ist definiert durch:

T Die von TaintDroid definierten Markierungen.

P Androide Berechtigungen.

A Die aktuell laufenden Apps.

C Die bestehenden Kommunikationsverbindungen.

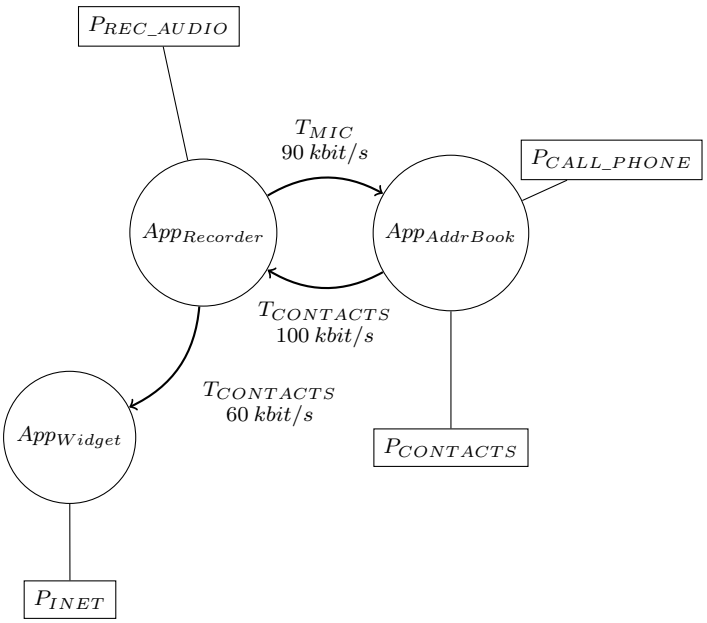


Abbildung 3.3.: Beispiel der dynamischen Systemdarstellung

$comm\,(c) \rightarrow (t, tp) : c \in C, t \in T, tp \in \mathbb{N}$ Die Parameter der Verbindungen.

$perm\,(a) \rightarrow P_a : a \in A, P_a \subset P$ Die Berechtigungen der Apps.

3.4.2. Knoten im Systemgraphen

Die Knoten im Systemgraph entsprechen den zurzeit laufenden Instanzen der DVM. Beim Starten einer App vom System wird diese dem Graphen hinzugefügt. Ausserdem werden, entsprechend der Informationen aus Androids PackageManager, die Rechte der entsprechenden App als Knoten $\boxed{P_\alpha}$ eingefügt.

Mehrere Prozesse, die unter der gleichen UID laufen, werden in einem Knoten zusammengefasst. Das wurde aus zwei Gründen gemacht:

1. Prozesse, welche in der gleichen Sandbox, d.h. mit der gleichen UID laufen, können über systemkernnahe Mechanismen, wie zum Beispiel Shared Memory, Daten austauschen. Dieser Kommunikationskanal wird von der hier entwickelten Erweiterung, wie auch bei XManDroid [Bugiel et al., 2011, S. 6], nicht einfach zu überwachen.

2. Für low-level Binder-IPC Nachrichten mit dem Typ FLAG_ONEWAY, steht auf Empfängerseite nur die UID des Senders und nicht die PID des Senders für Verifizierungszwecke zur Verfügung.

Bei Beenden einer App wird der dazugehörige Knoten und die mit Ihm verbundenen Berechtigungsknoten aus dem Graphen entfernt.

3.4.3. Darstellung der Informationsflüsse durch Kanten

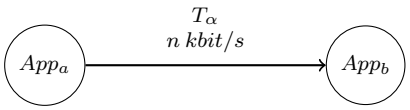


Abbildung 3.4.: Gewichtete gerichtete Kanten als Repräsentation der Kommunikation zwischen zwei Apps

Die Kommunikation zwischen den laufenden Instanzen der Apps wird als gewichtete gerichtete Kanten mit Bezeichnern in den Systemgraphen übertragen. Die Bezeichner (T_α) entsprechen der Klassifizierung der Daten, die auf diesem Kommunikationskanal übertragen wurden. Bei Nachrichten, die mit mehr als einer Quelle klassifiziert sind, wird jede entsprechende Kante in den Graphen eingefügt bzw. angepasst. Die Kantengewichte ($n\ kbit/s$) geben die aktuelle Bandbreite des Datentransfers auf dem Kommunkationskanal an. Die Nachrichten, welche über einen Kanal zwischen zwei Apps verschickt wurden, verlieren, nach definierter Zeit, ihren Einfluss auf das Gewicht (Bandbreite) des Kommunikationskanals.

3.5. Rechteausweitung im Systemgraphen

3.5.1. Definition

Eine mögliche Rechteausweitung wird, anders als bei dem Ansatz von Bugiel et al., nicht nur auf Basis der Rechte der Apps auf einem Pfad im Graphen definiert. Stattdessen werden die dynamischen Charakteristiken des Graphen, welche die Kommunikation in Echtzeit widerspiegelt, herangezogen.

Ein Datendurchsatzpfad ist ein Pfad im Systemgraphen, in dem alle Kanten den gleichen Bezeichner, d.h. die gleiche Klassifikation, haben.

3.5.2. Schwellenwert

Aufgrund der unterschiedlichen Eigenschaften der Markierungsquellen, bietet es sich an, diese Unterschiede auch bei der Erkennung von Rechteausweitung heranzuziehen. Wie schon in Abschnitt 3.1.3 erwähnt, haben Enck et al. die Markierungsquellen in Klassen eingeteilt. Besonders interessant sind hierbei die Klassen mit unterschiedlichen Bandbreiten. Ein Schwellwert für einen Informationsfluss von Geopositionen liegt wahrscheinlich niedriger, als einer für einen Informationsfluss, der Kamerabilder beinhaltet.

Eine Rechteausweitung tritt dann ein, wenn der Datendurchsatz auf einem Pfad im aktuellen Kommunikationsgraphen einen vorher definierten Schwellenwert überschreitet.

4. Implementierung

4.1. Einleitung

Das zuvor beschrieben Design ist als Erweiterung von Android 2.3 umgesetzt. Die Erweiterung setzt auf TaintDroid [Enck et al., 2010] um sensitive Daten im System zu markieren und zu propagieren. Für die Überwachung der Kommunikation wurde ein Systemdienst, im folgenden *FlowGraph* genannt, implementiert und mit der IPC-Middleware verbunden.

4.2. Entwicklungsumgebung

Der neuste Entwicklungszweig von Android, für den TaintDroid verfügbar ist, ist Android 2.3. Zur Entwicklung vom Android-System selbst in diesem Entwicklungszweig, nicht nur von Endnutzer-Apps, werden sowohl Ubuntu 10.04 als auch Mac OS X (Snow) Leopard offiziell unterstützt[1].

Das Build-Management von Android basiert auf Shellskripten, Makefiles und ist sowohl für den Buildprozess vom Systemkern, Userland und Standardapps des Android-Systems Zuständig. Außerdem werden auch auch die Entwicklungsutilities auf dem Hostsystem, wie z.B. adb[2], von ihm erstellt.

Es ist zu beachten, dass der sehr große Umfang an Quellcode des Android-Projekts auch erhebliche Anforderungen an ein Entwicklersystem und übliche IDEs stellt um effizient zu arbeiten.

4.3. TaintDroid

Um sensitive Daten in einem Android-System zu verfolgen und bei dem Verlassen des Systems, über die Netzwerkschnittstelle oder anderen Kommunikationsschnittstellen,

[1] `http://source.android.com/source/initializing.html`
[2] Android Debug Bridge

den Nutzer zu benachrichtigen, wurde von Enck et al. vor allem die Dalvik VM angepasst. Zur korrekten Propagierung der Wertmarkierungen bei den nativ implementierten Methoden des SDKs, wurden diese entsprechend angepasst. Dazu gehört unter Anderem Zugriff auf das Dateisystem oder auch Verschlüsselung. Damit Markierungen sich auch über das Dateisystem ausbreiten, ist es erforderlich einen Patch zur Unterstützung von erweiterten Attributen für die ext2- und Yaffs-Dateisysteme anzuwenden. Zuletzt wurden im Android-SDK Framework für die Markierungsquellen entsprechende Methoden angepasst, in denen Variablen mit sensitiven Daten initial markiert werden. Für Markierungssenken wurden die in Frage kommenden Methoden mit Logausgaben erweitert, die beim Verlassen von sensitiven Daten im System dieses Ereignis mitschneiden.

```
/
├── apps
├── bionic
├── bootable
├── build
├── cts
├── dalvik ..................... Taint-Propagierung innerhalb der Dalvik VM
├── development
├── device
├── external
├── frameworks
│   └── base ..................Taint-Quellen und Binder-IPC Taint-Propagierung
├── goldfish ......Unterstützung für erweiterte Attribute im YAFFS Dateisystem
├── hardware
├── libcore
│   ├── dalvik ................. Utility-Klasse für Zugriff auf TaintDroid-Strukturen
│   └── luni ................. Taint-Propagierung von/nach nativen Methoden des
│                              SDKs und Implementierung der Netzwerk-Senke für
│                              Markierungen
├── Makefile
├── ndk
├── packages
├── prebuilt
├── sdk
└── system
```

Abbildung 4.1.: Änderungen von TaintDroid im Android-System

In Abbildung 4.1 sieht man an welchen Stellen im Android-Quellcode, welche Änderungen von den TaintDroid-Entwicklern vorgenommen wurden.

4.4. FlowGraph

FlowGraph ist die zentrale Komponente der hier entwickelten Android-Erweiterung. Es ist ein zusätzlicher Systemdienst, welcher intern den aktuelle Systemzustand im Bezug auf die Kommunikation unter den Apps hält. Dazu gehören die laufenden Komponenten der Apps (Activities, Services, Broadcast Receivers und Content Provider) und deren aktuelle Kommunikation untereinander in Echtzeit.

4.4.1. Einordnung im System

Der FlowGraph-Dienst wird früh während des Boot-Prozesses vom Android-System gestartet. Der Dienst läuft innerhalb des *SystemServers*, wird in ihm vor anderen Diensten des Application Frameworks gestartet und unter einem eindeutigen Namen in der IPC-Middleware registriert. Vor allem wird FlowGraph vor dem *ActivityManagerService* gestartet, welcher unter anderem den Teil der IPC-Middleware enthält, welcher für die high-level Kommunikationsfunktionen wie Intents und Services zuständig ist.

Durch die Umsetzung als Service, ist die Erweiterung von allen Komponenten im System sehr einfach erreichbar. Dies wird vor allem mit der Binder-Middleware umgesetzt, die Prozesstransparenz unterstützt. Die Komponenten, welche mit dem FlowGraph-Dienst kommunizieren, müssen somit nicht wissen ob sich der Dienst im eigenen Prozess oder in einem anderen Prozess auf dem System befindet.

4.4.2. FlowGraph-Schnittstelle

Damit der FlowGraph-Dienst laufend einen dynamischen Kommunikationsgraphen in Echtzeit vorhalten kann, müssen bestehende Teile des Android-Framewokrs erweitert werden. Die Erweiterungen umfassen vor allem das Weitereichen von Ereignissen, wie Prozesserzeugung und -beendigung aber auch high-level Kommunikation zwischen diesen Prozessen.

Die Benachrichtigung über diese Ereignisse erfolgt via Binder-RPC. In Listing 4.1 ist die Schnittstellenbeschreibung in AIDL zu sehen, welche die Methoden auflistet, die vom FlowGraph-Dienst angeboten werden.

```
1 interface IFlowGraph
  {
3         void spawnProcess(int pid, int uid);
          void exitProcess(int pid, int uid);
5         void setProcessName(int pid, String name);

7         void preCommunication(int from_pid, int from_uid, int ↩
              to_pid, int to_uid, int size_in_bytes, int taint_tag);

9         String currentGraphState();
  }
```

Listing 4.1: Schnittstellendefinition von FlowGraph in AIDL

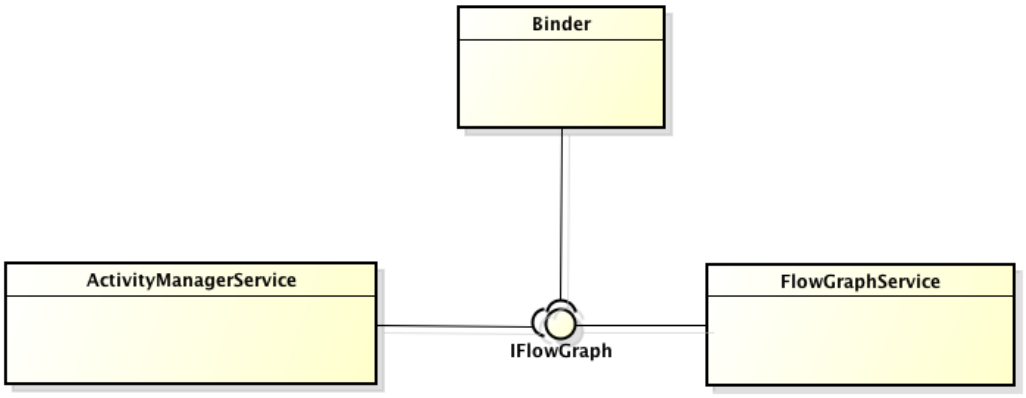


Abbildung 4.2.: Einbindung vom FlowGraph-Dienst ins bestehende Application-Framework

4.4.3. Überwachung der IPC-Kommunikation

Sämtliche Kommunikation zwischen Prozessen auf Android läuft über das Binder-IPC Framework, welches RPC-übliche Client- und Server-Stubs verwendet. An dieser Stelle setzt die hier entwickelte Erweiterung auch primär an, indem sie die empfangenen Nachrichten im Server-Stub abfängt und die relevanten Kommunikationsparameter an den FlowGraph-Dienst weiterleitet. Umgesetzt wurde dies in der *android.os.Binder.onTransact(...)*-Methode, die von jedem Binder Server-Stub verwendet wird. [Schreiber, 2011, S. 26]

Die Kommunikationsparameter, die an dem FlowGraph-Dienst weitergeleitet werden, setzten sich wie folgt zusammen:

1. PID des Initiators der Kommunikation
2. UID des Initiators
3. PID des Empfängers der Kommunikation
4. UID des Empfängers
5. Größe der empfangenen Nachricht in Bytes
6. Markierungslabel

Diese Parameter entsprechen auch denen der Signatur der *preCommunication*-Methode der IFlowGraph-Schnittstelle.

Es ist zu beachten, dass nach Beobachtungen, die wirkliche PID des Initiators, auf dieser low-level Ebene, bei Binder-IPC Nachrichten des Typs *FLAG_ONEWAY* nicht verfügbar ist.

Das Verfahren unterscheidet sich von bisherigen Ansätzen, wie zum Beispiel XManDroid [Bugiel et al., 2011] oder auch IPC Inspection[Felt et al., 2011b]. Bei Ihren Ansätzen werden nur die high-level Nachrichten überwacht und Berechtigungen der interagierenden Anwendungen überprüft. Besonders der Austausch von Nachrichten mit *Service*-Komponenten wird in den beiden Konzepten nicht genauer kontrolliert, sondern nur das Anfordern eines Client-Stubs für einen Dienst.

In der hier umgesetzten Implementation wird nur die Kommunikation von Endnutzer-Apps im Systemgraphen abgebildet.

4.4.3.1. Intents

High-level Nachrichten zwischen Apps, zu denen auch Intents gehören, werden auch mit Hilfe des low-level Binder ausgetauscht. Jedoch werden diese Nachrichten nicht direkt zwischen zwei kommunizierenden Anwendungen ausgetauscht, sondern werden über eine high-level Middleware weitergeleitet. Dieser Teil der Middleware verifiziert auch die Berechtigungen der potentiellen Empfängern eines Intents und führt ein Verzeichnis der registrierten Broadcast Receiver.

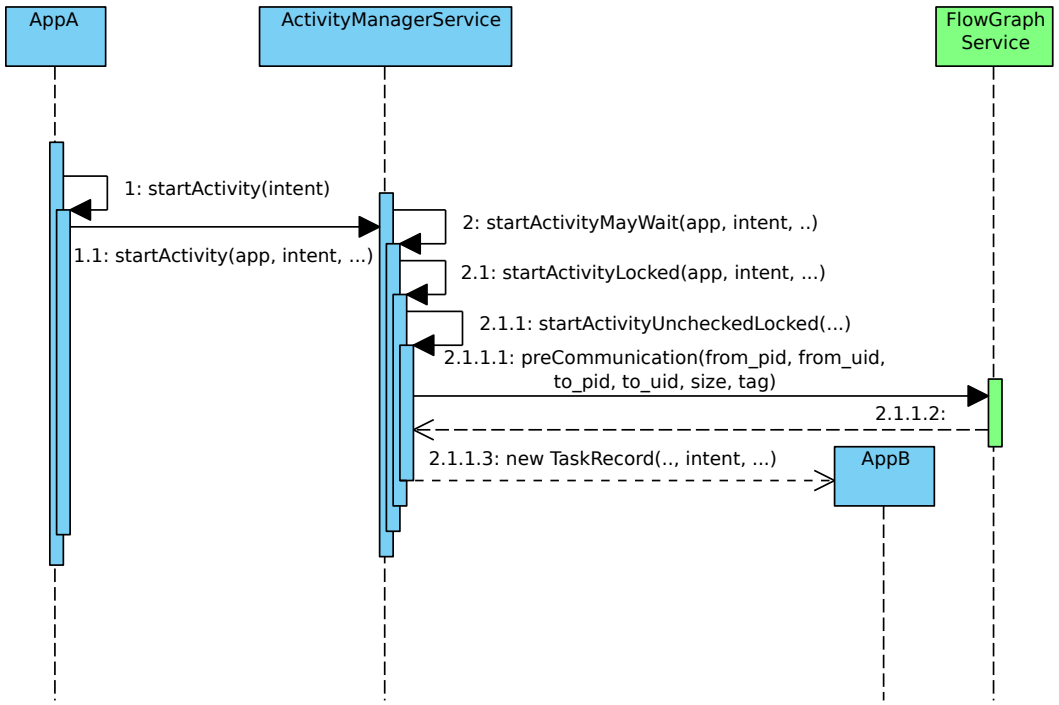


Abbildung 4.3.: Ablauf einer Kommunikation via Intents zwischen Apps

Der grobe Ablauf einer Intent-basierten Kommunikation zwischen zwei Apps auf einem Android-System ist in Abbildung 4.3 dargestellt.

Um dennoch die Kommunikation zwischen diesen zwei Apps korrekt im Systemgraphen abzubilden, wurde die high-level Middleware des Binder-Frameworks im *ActivityManagerService* und *ActivityStack* modifiziert.

Bei Intent-Nachrichten werden vor dem Starten einer neuen Activity-Komponente die relevanten Parameter der bevorstehende Kommunikation, mit Aufruf der *preCommunication*-Methode, an den *FlowGraph*-Dienst weitergeleitet. In Abbildung 4.3 wurde diese Weiterleitung der Kommunikationsparameter zwischen Schritt 2.1.1 und Schritt 2.1.1.3, in der Methode *ActivityStack.startActivityLocked(...)*, eingebaut.

4.4.3.2. Broadcast Intents

Die Auslieferung von *Broadcast Intents* wird in der *ActivityMangerServer.processCurBroadcastLocked(...)*-Methode umgesetzt. Für jeden Empfänger eines Broadcast Intents

wird eine Kante im Kommunikationsgraphen eingetragen bzw. aktualisiert, in dem die preCommuncation-Methode des FlowGraph-Dienst aufgerufen wird.

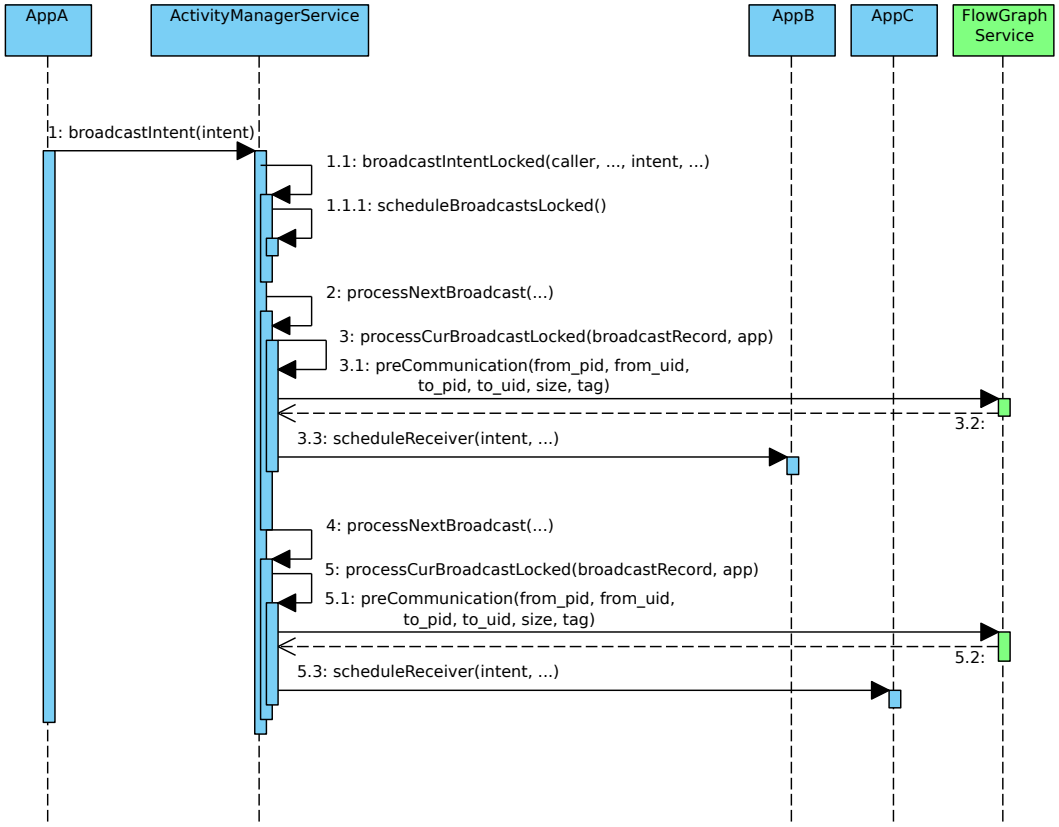


Abbildung 4.4.: Ablauf einer Kommunikation via Broadcast-Intents bei 2 Empfängern

Abbildung 4.4 zeigt das Versenden eines Broadcast-Intents von *AppA*. In diesem Beispielszenario sind zwei Apps (*AppB* und *AppC*) für diese Art von Broadcast-Intent registriert.

In Schritt 1.1.1 bestimmt die Binder-IPC-Middleware in Form des ActivityManagerService, welche Apps auf dem System Komponenten haben, die sich für die Art von Intents interessieren. Das Aussortieren geschieht via Intentfilter, welche in der Manifest-Datei einer App für jede BroadcastReceiver-Komponente spezifiziert werden können. Die bestimmten Empfänger der Intent-Nachricht werden zu einer internen Liste hinzugefügt.

Schritt 2 und 4 beschreiben wie für die einzelnen Empfangskomponenten aus der internen Liste, die Intent-Nachricht an die eigentlichen Komponenten ausgeliefert werden. Innerhalb von der *processNextBroadcast()*-Methode wird vor Auslieferung überprüft, ob

die empfangende App ausreichende Berechtigungen besitzt.

In Abbildung 4.4, werden zwischen Schritt 3 (resp. 5) und Schritt 3.3 (resp. 5.3) die relevanten Kommunikationsparameter an den *FlowGraph*-Dienst weitergeleitet.

4.4.3.3. Services

Kommunikation zwischen Apps, bei denen eine Service-Komponente involviert ist, werden über die Modifikationen der *android.os.Binder*-Klasse überwacht. Wenn ein Service von einer anderen App gebunden wird, wird ein Server-Stub vom Binder-Framework erstellt und bei Bedarf ein neuer DVM Prozess erstellt. Die Parameter jeder einzelne Kommunikation zwischen einer App und dem Service werden an den FlowGraph-Dienst weitergeleitet, welcher sie in den dynamischen Informationsflussgraphen integriert.

Der detaillierte Ablauf, einer Binder-RPC Kommunikation, ist in Abbildung 4.5 zu sehen. In der selben Abbildung ist auch die, aus anderen RPC-Umsetzungen, wie Java RMI oder CORBA, bekannte Klassenhierarchie mit Client-Proxy und Server-Stub zu erkennen.

Innerhalb der *Binder.execTransact(...)*-Methode werden jeweils vor und nach dem Aufruf der *onTransact()*-Methode die relevanten Kommunikationsparameter an den *FlowGraph*-Dienst weitergereicht. Somit wird sowohl der Hin-, als auch der Rückkanal der Kommunikation vom System überwacht.

4.4.3.4. ContentProvider

ContentProvider stellen eine zentrale Komponente vieler Apps dar und können auch mit mehreren Apps geteilt werden. Deren Schnittstelle bietet grundlegende CRUD-Funktionalitäten [Google, 2012] und wird mit Hilfe des, unter Android üblichen, Binder-RPC Mechanismus zugegriffen. Somit kommen hier auch die Erweiterungen, welche in Kapitel 4.4.3.3 beschrieben wurden, zum Tragen.

Bevor Aufrufe auf einen ContentProvider ausgeführt werden können, muss die App sich ein Proxy-Objekt für die gewünschte ContentProvider-Instanz beschaffen. Für diesen Zweck bietet der *ActivityManagerService* die *getContentProvider()*-Methode an, der beim Aufruf eine Referenz zum eigenen Thread und der Name des gewünschten ContentProviders übergeben wird. Dieser Aufruf ist sowohl am Anfang von Abbildung 4.6 zu sehen also auch in Abbildung 4.7.

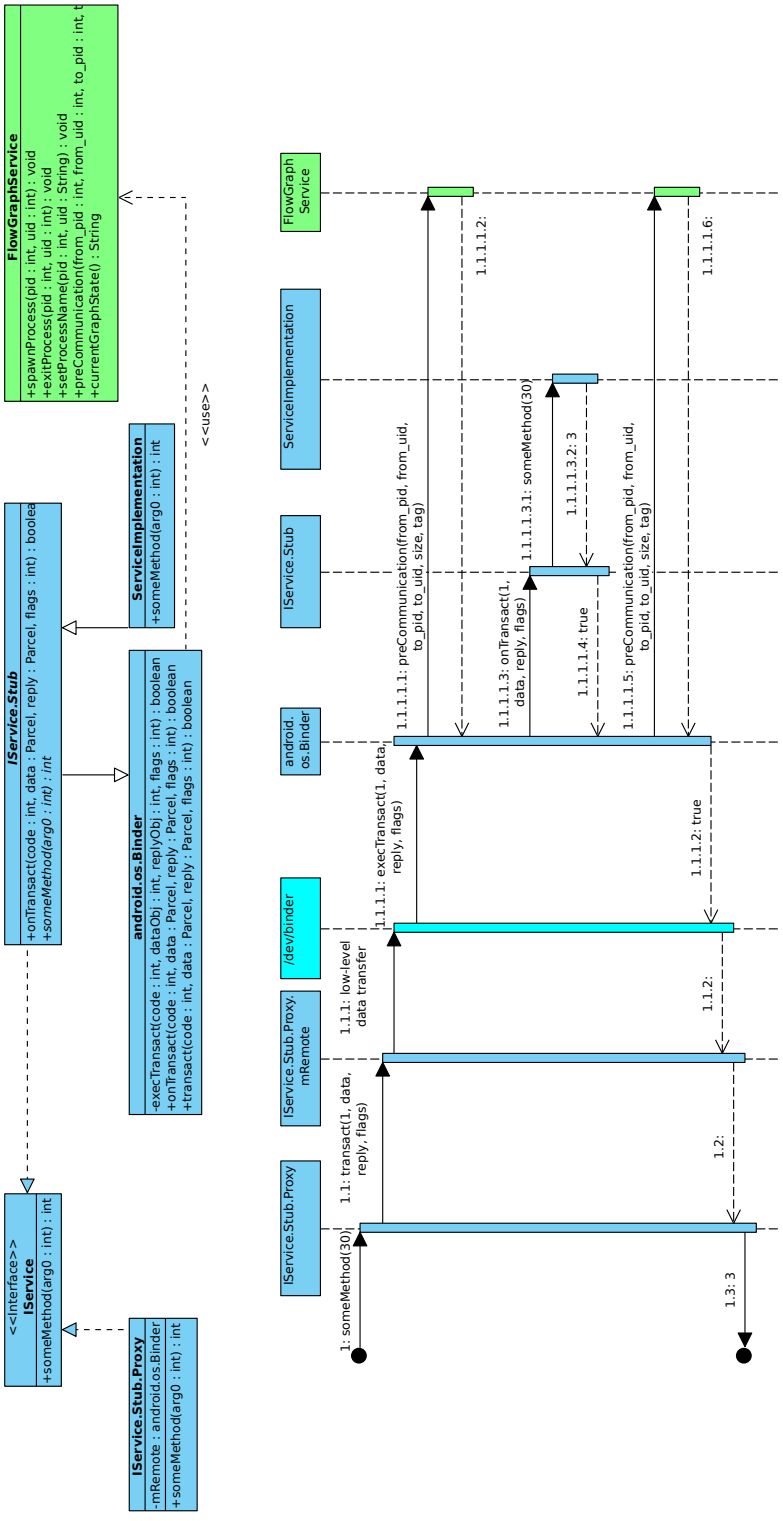


Abbildung 4.5.: UML Klassen-/Sequenzdiagramm zur Darstellung der Struktur und des Ablaufs der Binder-RPC Kommunikation

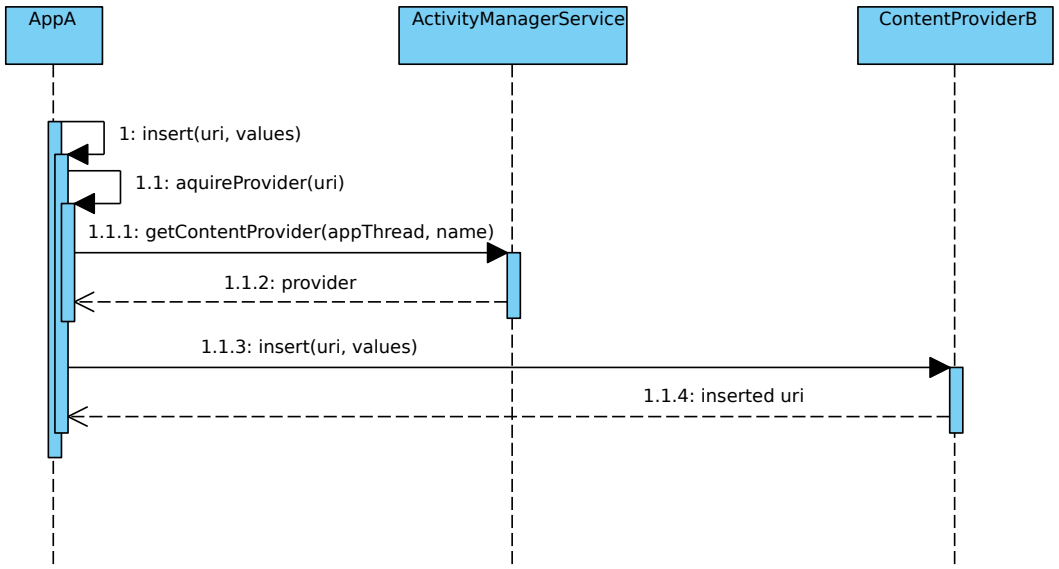


Abbildung 4.6.: UML Sequenzdiagramm zur Darstellung des Ablaufs eines *insert()*-Aufrufs auf einen ContentProvider

Um Einträge in einen ContentProvider hinzuzufügen oder zu ändern, gibt es die Methoden *insert()* und *update()* in der allgemeinen Schnittstelle von ContentProvidern. Da sich das Einfügen neuer Einträge und das Ändern bestehender Einträge vom Ablauf nicht nennenswert für unsere Implementierung unterscheidet, wird im Folgenden nur auf das Einfügen neuer Einträge eingegangen. Dieser Aufruf ist vereinfacht in Abbildung 4.6 zu sehen.

In Schritt 1.1.3 in Abbildung 4.6 werden mit Binder-RPC neue Einträge in den ContentProvider einer anderen App eingefügt. Die neuen Einträge, unter umständen auch markiert, stehen im Parameter *Values* und werden beim RPC-Aufruf serialisiert und übertragen. Diese Markierung und die restlichen Kommunikationsparameter des *insert()*-Aufrufs werden im Schritt 1.1.1.1.1 in Abbildung 4.5 an den *FlowGraph*-Dienst weitergeleitet.

Die Abfrage von Daten aus einer ContentProvider-Komponente ist wesentlich komplexer. Hierfür gibt es die *query()*-Methode in der ContentProvider-Schnittstelle.

Ein vereinfachter Aufruf ist in Abbildung 4.7 dargestellt. Um hier eine Überwachung der Kommunikation zu ermöglichen wurde die *CursorWindow*-Klasse des Android

Frameworks angepasst werden.

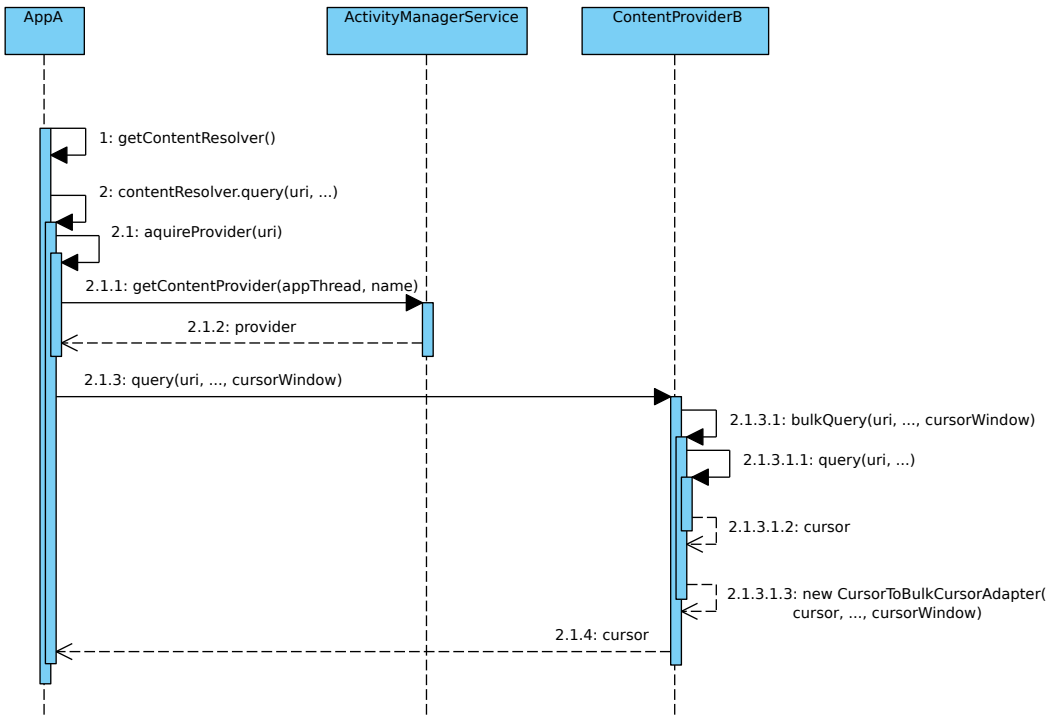


Abbildung 4.7.: UML Sequenzdiagramm zur Darstellung des Ablaufs eines *query()*-Aufrufs auf einen ContentProvider

Nachdem in Schritt 2.1.3.1 in Abbildung 4.7 der ContentProvider die Abfrage erhalten hat, wird diese an die *query()*-Methode weitergeleitet, welche einen Cursor auf das Ergebnis zurückliefert. Die Cursor-Klasse ist eine Umsetzung des verbreiteten Iterator-Entwurfsmusters [Gamma et al., 1995, S. 257ff.].

In Schritt 2.1.3.1.3 wird eine spezielle Implementierung des Cursors erstellt und letztendlich durch Binder-RPC wieder zurück, an die letztendlich aufrufende App, gegeben. Dieser spezielle Cursor, in Form der *CursorToBulkCursorAdapter*-Klasse, bekommt bei Initialisierung unter anderem das, von der aufrufenden App in Schritt 2.1.3 übergebene, *CursorWindow* und den zuvor in Schritt 2.1.3.1.2 erhaltenen Cursor übergeben.

Die *CursorWindow*-Klasse dient dem Puffern eines Teilausschnitts der Ergebnistabelle und ist intern nativ in C++ implementiert. Hiermit wird die Binder-IPC Kommunikation zwischen App und ContentProvider reduziert, da nicht mit jedem Zugriff auf den

	1	2	3	...
1	0	$T_{CONTACTS}$	0	0
2	$T_{HISTORY} \vee T_{SMS}$	0	T_{MIC}	0
3	0	0	0	0
⋮	0	0	0	0

Tabelle 4.1.: Beispiel für 2D-Array zur Darstellung von Markierungen der Zellen innerhalb einer *CursorWindow*-Klasse

Cursor-Proxy eine weitere Anfrage an den ContentProvider geschickt werden muss. Da TaintDroid die Verbreitung von Markierungen von Variablen nur in DVM-Bytecode unterstützt und nicht für nativen Maschinencode, wurde die Klasse um ein zweidimensionales Array für die Markierungen der jeweiligen Zellen erweitert. Auf diese Weise werden die Markierungen der Daten innerhalb des ContentProviders in die aufrufende App propagiert und dort bei Weiterverarbeitung weiter verbreitet. Viel wichtiger ist jedoch, dass jetzt bei der Serialisierung des CursorWindows die Markierungen der Orginaldaten des ContentProviders zur Verfügung stehen und somit auch an den *FlowGraph*-Dienst weitergeleitet werden können.

In Tabelle 4.1 sieht man ein Beispiel für die neu hinzugefügte 2D-Array für die Markierungen. Bei jedem Schreibzugriffs auf das CursorWindow wird die Markierung der beschriebenen Zelle im 2D-Array mitgespeichert. Beim Lesezugriff werden angeforderte Zellenwerte neu mit den Orginalmarkierungen markiert und dann zurückgegeben.

Das empfohlene Backend zum permanenten Speichern der Daten eines ContentProviders ist SQLite[3]. Um jedoch eine Verbreitung von Markierungen nach und innerhalb SQLite zu ermöglichen, wären weitaus umfangreichere Anpassungen notwendig, ohne die Markierungen von Daten beim Schreiben in SQLite Datenbanken verloren gehen.

Durch die Anpassung des CursorWindows wurde die Grundlange für weitergehende Ausbreitung von Markierungen geschaffen. TaintDroid selbst markiert Zugriffe auf spezielle ContentProvider wie zum Beispiel Adressbuch, Webbrowserverlauf und SMS/MMS-Speicher.

[3] https://www.sqlite.org/

4.4.4. FlowGraph-Dienst Implementation

Der FlowGraph-Dienst ist in Java umgesetzt und wird früh in der Bootsequenz gestartet, damit die Überwachung der Kommunikationskanäle auch alle Endnutzer-Applikationen umfasst. Intern hält FlowGraph laufend den Zustand der aktuellen Kommunikation zwischen Apps. Konkret umfasst der interne Zustand von FlowGraph folgendes:

- die zurzeit laufenden Apps welche unter ihrer UID-Sandbox zusammengefasst werden
- die gerichteten Kommunikationskanäle zwischen zwei Apps bestehend aus:
  - die Quelle- und Zielsandbox des Kommunikationkanals
  - die Markierung der Kommunikationkanals
  - die übertragene Datenmenge im vorherigen Zeitintervall von $\Delta t_{monitor}$

Für $\Delta t_{monitor}$ wurde eine Zeit von $60s$ gewählt, welche fließend für jeden Kommunikationskanal im Systemzustand aktuell gehalten wird.

4.4.4.1. Aktueller Datendurchsatz eines Kommunikationskanals

Der laufend überwachte Zeitintervall von $60s$ wurde so gewählt, das Schwankungen in der Ausführung der Programme die Messung nicht zu stark beeinträchtigen. Ausserdem soll damit verhindert werden, das böswillige Apps Daten auf lange Zeit Zwischenspeichern, um sie danach in geringerer Bandbreite auszutauschen.
Ein zu kleiner Zeitraum für $\Delta t_{monitor}$ hätte zur Folge, dass man weniger Zusammenhänge zwischen verschiedenen Apps schließen kann, da alte Daten sehr schnell aus der Repräsentation der Systemkommunikation verschwinden. Ist der gewählte Zeitraum für $\Delta t_{monitor}$ jedoch zu groß, würde der Systemgraph laufend wachsen und die darauf angewendeten Algorithmen unweigerlich verlangsamen.

Es werden nur Daten für eine Kommunikationskanal zwischen zwei Apps gezählt, wenn diese zuvor von TaintDroid markiert wurden. Die Markierung und die Datenmenge werden in der jeweiligen Kante für die Kommunikation zwischen zwei Apps gespeichert. Wenn für den Zeit von einer Minute keine Kommunikation über einen Kanal stattgefunden hat, wird diese Kante aus dem Graphen des Systemzustands entfernt.

4.5. Visualisierung vom FlowGraph-Zustand

Die aktuell überwachten Kommunikationen werden laufend innerhalb des FlowGraph-Dienst vorgehalten, und über Zeit durch weiterer Kommunikation im System angepasst. Bis auf Logausgaben, gibt es keine Möglichkeit auf den aktuellen Systemzustandsgraphen zuzugreifen.

Um zu Studienzwecken dennoch eine Visualisierung zu ermöglichen, wurde das `flowgdmp`-Kommandozeilenprogramm entwickelt. Mit Hilfe von `flowgdmp` und einem Visualisierungsprogramm für Graphen, wie zum Beispiel Graphviz[4], kann der aktuelle Zustand grafisch dargestellt werden.

4.5.1. `flowgdmp` Kommandozeilenprogramm

Angelehnt an das `service`-Kommandozeilenprogramm, welches unter `/system/bin/service` auf einem Android-System zu finden ist, wurde das Programm *flowgdmp* entwickelt. Das Programm ist in C++ geschrieben und ruft mit Hilfe von Binder-RPC die *currentGraphState()*-Methode des FlowGraph-Dienst auf, um den aktuellen Systemgraphen im DOT Format zu erhalten.

Von einem Entwicklungsrechner, welcher Zugang zu einem Android-System im Emulator oder via USB zu einem echten Gerät hat, kann `flowgdmp` via `adb shell` aufgerufen werden um den aktuellen Systemzustand zu erhalten.

4.5.2. `flowgsnapshot.sh` Skript

Darüber hinaus wurde ein Hilfsskript für die Bash-Shell[5] entwickelt, welches die kontinuierliche Überwachung eines Android-Systems zu Test- und Evaluationszwecken erleichtert. Dessen Verwendung ist im folgenden Logausschnitt kurz beschrieben.

```
tdroid:~/tdroid$ ./flowgsnapshot.sh 5 run_no_1
Press [CTRL+C] to stop..
Press [CTRL+C] to stop..
Press [CTRL+C] to stop..
^C
```

Im Logausschnitt wurden bis zur Beendigung alle 5 Sekunden ein Schnappschuss des FlowGraph-Zustands gemacht und in den Ordner `run_no_1` gespeichert.

[4] `http://graphviz.org`
[5] `http://www.gnu.org/software/bash/`

4.5.3. Graphviz Visualisierung

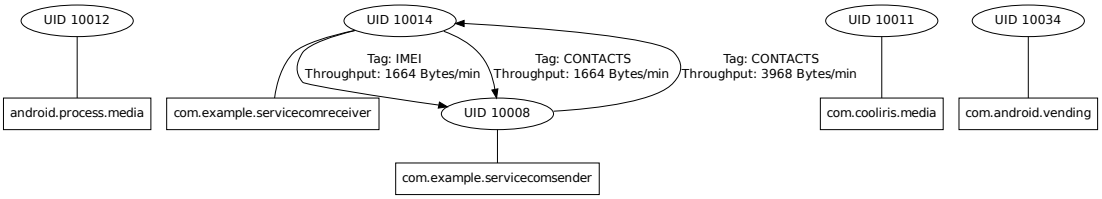


Abbildung 4.8.: Ausschnitt eines Systemzustand-Schnappschuss via `flowgdmp` nach Anwendung von Graphviz

Mit `flowgdmp` in Kombination mit Graphviz kann man somit ein grafisches Abbild des aktuellen Systemzustands bekommen. Folgend ist ein Beispielkommando aufgeführt mit der die Grafik, die in Abbildung 4.8 zu sehen ist, generiert wurde:

```
adb shell flowgdmp | dot -Gaspect=1 -Kdot -GK=4 -Tpdf  \
-osnapshot.pdf
```

Abbildung 4.8 spiegelt ein Szenario wieder, bei der die App `com.example.service-comsender` Kontaktdaten an `com.example.servicecomreceiver` gesendet hat und diese App mit den empfangenen Daten zusätzlich angehängter IMEI antwortet.

5. Evaluation

5.1. Anforderungen

In der folgenden Evaluation soll das hier entworfene Design und im Speziellen die hier entwickelte Umsetzung dieses auf die im Anfang angesprochenen Ziele hin getestet werden. Insbesondere geht die Evaluation auf die nachfolgenden Ziele für die entwickelte Android-Erweiterung ein:

- Erkennung und Abwehr von durch Kommunikation entstandene Rechteausweitung
- Speziell die Ausweitung von Rechten die im Zusammenhang mit sensitiven Daten stehen
- Erkennung von Rechteausweitung durch konspirierende Apps (*colluding applications*) und durch fehleranfällige Apps (*confused-deputy*)
- Entwicklung von Endnutzer-Apps ist unverändert im Vergleich zum offiziellen Android
- Funktionalität von harmlosen Apps wird nicht beeinträchtigt

5.2. Ansatz

Um die Android-Erweiterung auf die eben benannten Ziele hin zu überprüfen werden zwei verschiedene Szenarien vorgestellt und umgesetzt. Jedes Szenario beschreibt eine bestimmte Anzahl von Apps, deren Berechtigungen und wie diese Apps miteinander interagieren.

Die hier vorgestellten und entwickelten Apps dienen der Evaluation der Android-Erweiterung und sind weder auf Benutzerfreundlichkeit noch auf Vollständigkeit hin entwickelt. Es wurde primär ein Fokus auf die für die Kommunikation relevanten Programmteile gelegt.

5.2.1. Szenario A

Diese Szenario zielt darauf ab, die Android-Erweiterung auf Erkennung von Rechteausweitung durch konspirierende Apps zu überprüfen.

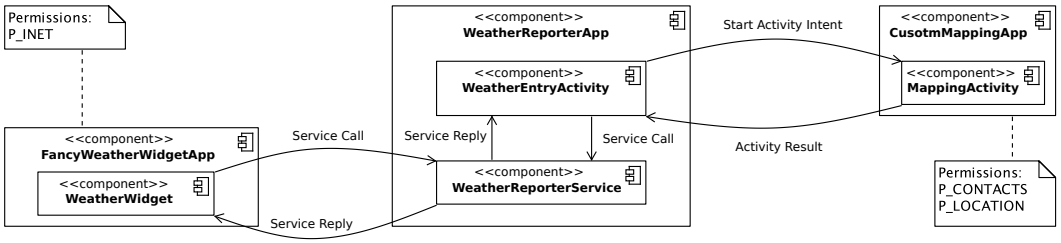


Abbildung 5.1.: Szenario A: Apps, deren Komponenten und deren Interaktion untereinander

Bestandteil dieses Szenarios sind die drei Apps *FancyWeatherWidgetApp*, *WeatherReporterApp* und *CustomMappingApp*. Alle Apps sind von verschiedenen Entwickeln, d.h. durch unterschiedliche Entwicklerschlüssel signiert, und laufen somit als jeweils unterschiedliche Systemnutzer.

FancyWeatherWidgetApp stellt eine App Widget dar, welches auf dem Startbildschirm von Android laufend eigene Wettereinträge oder Einträge aus dem Internet darstellt. Hierfür fragt es alle 30 Sekunden die vorhanden Daten von der *WeatherReporterApp* und bekommt dabei vom Nutzer unbemerkt auch Kontaktdaten. *WeatherReporterApp* dient der Eingabe eigener Wettereinträge, hält diese vor und bietet diese mit einer Service-Komponente anderen Apps an. Schließlich gibt es die *CustomMappingApp*, womit auf einer Karte eine Position ausgewählt werden kann. Die Karte enthält die eigene aktuelle Geoposition und die Adressen der Kontakte. Diese App liefert zusätzlich zur Position auch versteckt Kontaktdaten zurück. Diese werden im *WeatherReporterService* angesammelt, um auf Anfrage vom *FancyWeatherWidget* ausgeliefert werden zu können.

Es wird davon ausgegangen, dass die Android-Erweiterung einen zu hohen Informationsfluss an Kontaktdaten zwischen *WeatherReporterService* und *FancyWeatherWidget* erkennt und diesen unterbindet.

Für diese Szenario wurde für Kommunikationskanten mit der Markierung $T_{CONTACTS}$ ein Schwellenwert von $1000 \frac{Bytes}{min}$ gewählt.

In Abbildung 5.1 sieht man die Interaktion dieser Apps und deren besonderen Berechtigungen.

5.2.2. Szenario B

In diesem Szenario wird der FlowGraph-Dienst auf die Erkennung von Rechteausweitung durch Missbrauch von Schnittstellen überprüft.

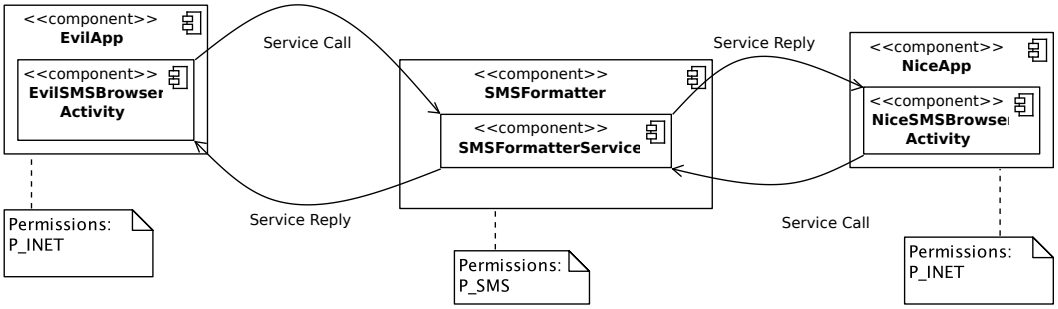


Abbildung 5.2.: Szenario B: Apps, deren Komponenten und deren Interaktion untereinander

Zentral für diese Szenario ist der *SMSFormatterService*, welcher auf Anfrage von anderen Apps, eine in HTML vorformatierte SMS zurückgibt. Der Dienst wurde naiv umgesetzt und führt keine weitergehende Überprüfungen auf Berechtigungen der aufrufenden Apps durch. Hierzu hat er die Berechtigung auf die eingegangenen SMS im System zuzugreifen.

Zu diesem Dienst gibt es noch die gutmütige *NiceApp*. In dieser kann der Nutzer eine Nachrichtennummer eingeben und daraufhin wird diese Nachricht vom *SMSFormatterService* angefordert. Die zurückgegebene formatierte Nachricht wird in der eigenen Activity angezeigt.

Schließlich gibt es noch die sich widerrechtlich verhaltende Anwendung *EvilApp*. Von der Benutzeroberfläche ist sie mit der *NiceApp* identisch. Obendrein ist auch das vom Nutzer zu beobachtende Verhalten der beiden Apps exakt gleich. Die *EvilApp* fragt jedoch alle SMS die es bekommen kann ab, stellt jedoch nur die vom Nutzer angeforderte Nachricht dar. Im Hintergrund werden alle Nachrichten ins Internet versendet.

An dieser Stelle wird erwartet, dass der FlowGraph-Dienst den bevorstehenden unnatürlich hohen Informationsfluss von SMS-Daten zwischen *SMSFormatterService* und

EvilApp bemerkt und verhindert.

Der Schwellenwert für Kommunikationskanten mit der Markierung T_{SMS} wurde auf $10000 \frac{Bytes}{min}$ festgelegt.

Abbildung 5.2 zeigt die Apps dieses Szenarios, deren Berechtigungen und Komponenten. Ausserdem ist die erwartete Kommunikation zwischen den Apps dargestellt.

5.3. Durchführung der Evaluation

Zur Durchführung wurden die Szenarien manuell durchgeführt und mit `flowgsnapshot.sh` überwacht. Nachfolgend sind die relevanten Zustandsschnappschüsse des Testablaufs dargestellt. Die in Abbildung 5.3 und Abbildung 5.4 aufgelisteten Zustandsschnappschüsse sind in chronologischer Reihenfolge aufgeführt.

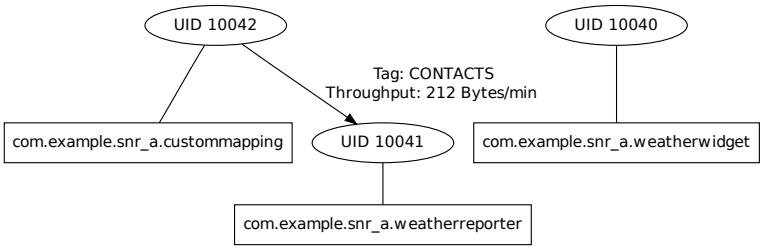
5.3.1. Szenario A

Folgend ist eine kurze Beschreibung der Ereignisse rund um die Schnappschüsse in Abbildung 5.3:

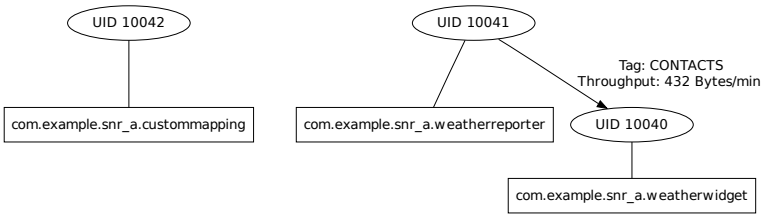
Zeitpunkt 1 Der Benutzer hat die *WeahterReporterApp* gestartet um einen neuen Eintrag anzulegen. Um die Geoposition auszuwählen wurde die *MappingActivity* gestartet. Diese liefert nicht nur die ausgewählte Position zurück, sondern auch einen Eintrag aus dem Adressbuch.

Zeitpunkt 2 Das *WeatherWidget* ruft alle 30 Sekunden die im *WeatherReporterService* vorhandenen Wettereinträge ab. Hierbei werden auch, vom Nutzer unbemerkt, zwischengespeicherte Kontaktdaten mit abgerufen. Anfangs sind nur wenig Kontaktdaten zwischengespeichert, womit nur ein geringer Informationsfluss erreicht wird.

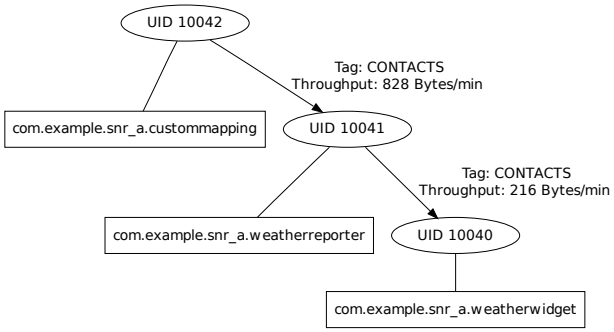
Zeitpunkt 3 Vom Benutzer werden weiterhin neue Wettereinträge angelegt und dabei ganz natürlich mit der *MappingActivity* weiterhin Kontaktdaten unbemerkt an den *WeatherReportService* weitergeleitet. Dadurch werden die im *WeatherReportService* zwischengespeicherten Kontaktdaten immer mehr.



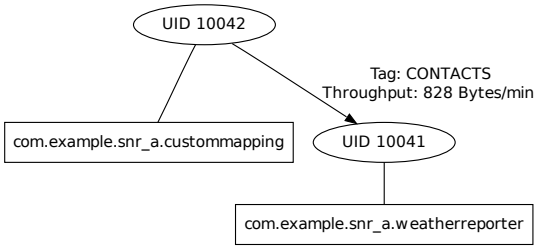
(a) Zeitpunkt 1



(b) Zeitpunkt 2



(c) Zeitpunkt 3



(d) Zeitpunkt 4

Abbildung 5.3.: Szenario A: Zustandsschnappschüsse

Zeitpunkt 4 Bei einer erneuten Abfrage vom *WeatherWidget* stehen im *WeatherReportService* wesentlich mehr Kontaktdaten zur Verfügung. `com.example.snr_a.weatherwidget` wurde vom FlowGraph-Dienst beendet da eine Kommunikation versucht wurde, bei der die Verbindung zwischen `com.example.snr_a.weatherreporter` und dem App-Widget mehr als den zuvor definierten Schwellenwert von $1000\frac{Bytes}{min}$ erreicht hätte. Dieser Kommunikationsversuch stellt den Antwortsversuch von *WeatherReportService* auf die Anfrage vom Widget dar.

5.3.2. Szenario B

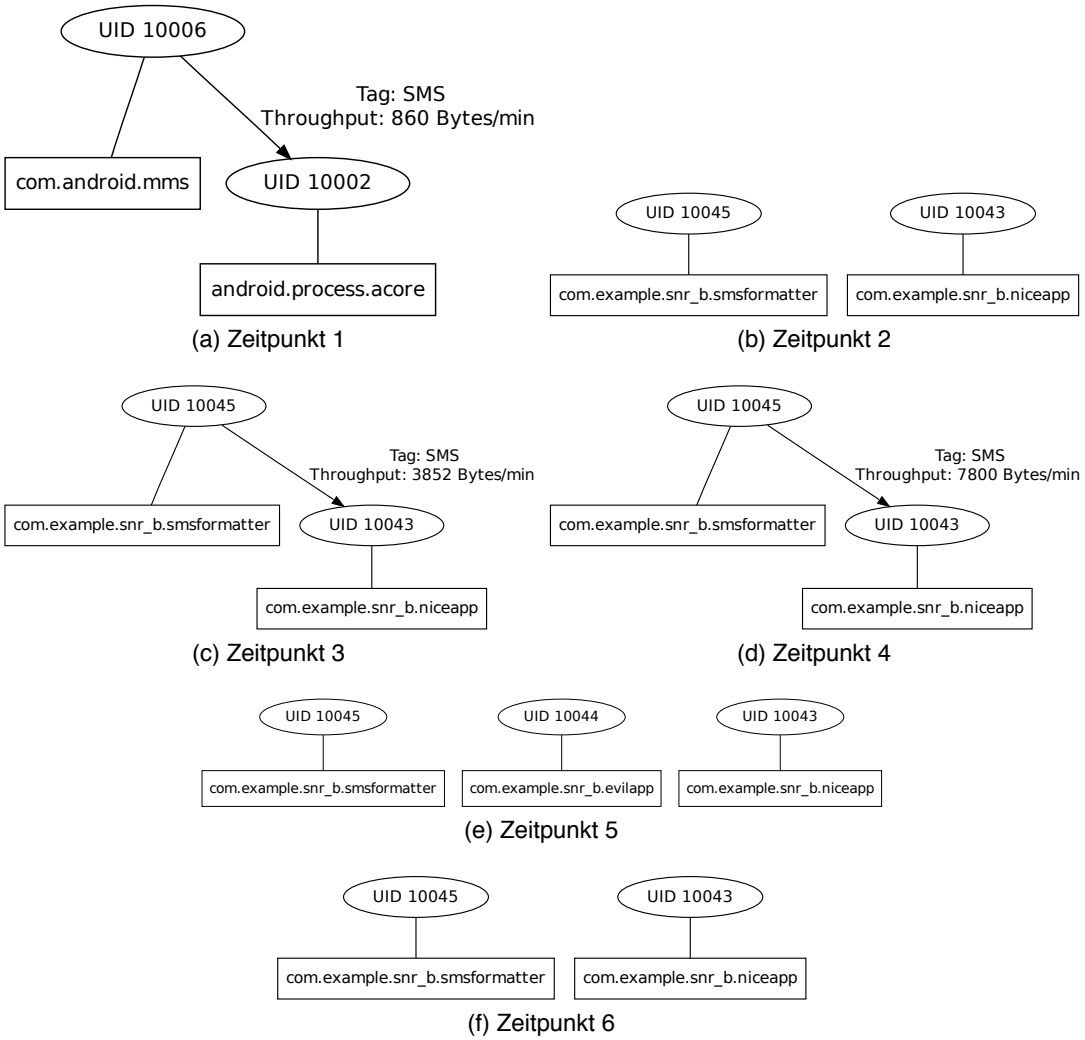


Abbildung 5.4.: Szenario B: Zustandsschnappschüsse

Eine Beschreibung der interessanten Schnappschüsse aus Abbildung 5.4:

Zeitpunkt 1 Vor dem Start des eigentlichen Szenarios ist eine Kommunikation zwischen der standard Messaging-App (`com.android.mms`) und des Systemprozesses `android.process.acore`.

Zeitpunkt 2 Die *NiceApp* (`com.example.snr_b.niceapp`) wurde vom Nutzer gestartet, welche beim Start ein Proxy des *SMSFormatterService* (`com.example.snr_b.smsformatter`) bindet. Hierbei wird der für den Dienst benötigte Prozess gestartet.

Zeitpunkt 3 Der Nutzer fragt in der *NiceApp*, durch Tippen auf einen Button, das Anzeigen einer SMS an. Dazu wird via Binder-RPC eine formatierte SMS von dem *SMSFormatterService* abgefragt.

Zeitpunkt 4 Wie zuvor, fragt der Nutzer erneut eine Nachricht zur Anzeige in der *NiceApp* an.

Zeitpunkt 5 Die *EvilApp* (`com.example.snr_b.evilapp`) wird vom Nutzer gestartet.

Zeitpunkt 6 Der Nutzer möchte eine Nachricht zur Darstellung anzeigen. Jedoch fragt die bösartige *EvilApp* alle Nachrichten von *SMSFormatterService* ab. Danach würde es nur die angeforderte Nachricht darstellen und die anderen Nachrichten über das Internet schicken. Hierzu kommt es jedoch nicht, da die *EvilApp* zuvor von dem *FlowGraph*-Dienst beendet wird, da der exzessive Zugriff auf die SMS-Nachrichtenbox einen Missbrauch darstellt.

5.4. Beobachtungen und Ergebnis

5.4.1. Grundlegende Beobachtungen

Die Rechteausweitungen, sowohl die basierend auf drei konspirierenden Apps, als auch die mittels unkontrollierter öffentlicher Schnittstelle, wurden von der Android-Erweiterung erkannt und unterbunden. Dabei wurde eine legitime Nutzung der unkontrollierten Schnittstelle aus Szenario B zugelassen. Andere installierte Apps, wie die standard Messaging-App oder das Adressbuch, sind unverändert funktionsfähig.

Beide Testanwendungen für Rechteausweitung versuchen sensitive Daten, über die berechtigten Sandboxen hinaus, zu verbreiten. Unabhängig von den Berechtigungen

der bösartigen Apps wurde übermäßige Ausbreitung von sensitiven Daten erkannt und verhindert.

Des Weiteren wurden die Apps aus Szenario A und Szenario B gegen die Android 2.3.2 API entwickelt. In den Anwendungen selbst wurden keine, für die Android-Erweiterung, speziellen Funktionen verwendet. Die Apps laufen problemlos auf Android-Systemen, welche die hier entwickelte Erweiterung nicht beinhalten.

Wie in Abbildung 5.4a zu sehen, wurde nicht nur die Kommunikation zwischen den zur den Szenarien gehörenden Apps beobachtet, sondern auch Kommunikation zwischen Standard-Apps, die sich auf fast jedem Android-System befinden, und Systemdiensten.

Außerdem ist auffällig, dass, bei Verwendung der ContentProvider-API, schnell größere Datenmengen über kurze Zeitspannen auf den Kommunikationswegen entstehen können. Grund hierfür ist vermutlich, dass von dem Cursor intern verwendete und in Abschnitt 4.4.3.4 beschriebene, CursorWindow. Dieses wird bei der Rückgabe des Curors an den aufrufenden Code mutübertragen und beinhaltet eine größere Teilmenge des Ergebnisses, um die Anzahl der Kommunikationsinteraktionen zwischen aufrufenden Code und des ContentProviders zu verringern.

5.4.2. Auswahl passender Schwellenwerte

Bei der Entwicklung und Durchführung der Evaluation ist Aufgefallen, dass die Auswahl eines passenden Schwellenwerts für die korrekte Funktion der Android-Erweiterung ausschlaggebend ist.

Für die Durchführung der beiden Szenarien wurden beim Testen die Kommunikationsverbindungen überwacht und mögliche Richtwerte für legitime Kommunikation zwischen Apps ermittelt. Für Daten mit der Markierung $T_{CONTACTS}$ wurde dieser auf $1000\frac{Bytes}{min}$ festgelegt, da die Übertragung von einem Adressbucheintrag circa 300 Bytes beansprucht. Bei Daten mit der Markierung T_{SMS} wurde der Schwellenwert auf, die im Vergleich zu $T_{CONTACTS}$ wesentlich größeren, $10000\frac{Bytes}{min}$ gesetzt. Alleine das Auslesen einer SMS verursacht einen Datenaustausch zwischen zwei Apps von mehr als 2000 Bytes.

5.4.3. Ergebnis

Die Beobachtungen beider Szenarien zeigt, dass das Design zur Darstellung von dynamischen Informationsflüssen und die hier vorgestellte Implementierung für die Android-Plattform, erfolgreich Rechteausweitung erkennen und unterbinden.

Wobei die hier verwendeten Szenarien explizit mit dem Wissen über die Existenz der Android-Erweiterung entwickelt wurden, stellen sie dennoch realistische Apps dar. Mit dem Wissen von der Existenz der Erweiterung sind aber auch bestimmte Methoden vorstellbar, welche die Erkennung der Rechteausweitung verhindern. Dazu gehören zum einen ein allgemeiner Ansatz, das komprimieren von den sensitiven Daten vor der Übertragung, oder aber auch das Puffern der sensitiven Daten auf Senderseite, um mit einem geringen, aber dafür konstanten, Informationsfluss Daten unter den definierten Schwellwerten zu übertragen.

6. Fazit

Im Zuge der Arbeit wurde sowohl ein Design für die dynamische Darstellung der Interprozesskommunikation auf einem Android-System erstellt, als auch eine Referenzimplementierung dieses Designs für die Android 2.3 Plattform entwickelt. Hierfür wurde auf die vorhandene Arbeit des TaintDroid-Projekts aufgebaut. Die Überwachung der verschieden Kommunikationskanäle auf Android stellte sich unterschiedlich aufwändig dar und insbesondere die Anpassung der ContentProvider-API war sehr zeitintensiv und konnte nur unvollständig angepasst werden.

Es wurde gezeigt, dass es Möglich ist, mittels der dynamischer Darstellung von Informationsflüssen, eine Ausbreitung von sensitiven Daten, verursacht durch die Ausweitung von Rechten, erfolgreich zu erkennen und zu unterbinden. Gerade die Evaluation auf Basis von Test-Szenarien hat dies bestätigt.

Jedoch hängt die Zuverlässigkeit der Erkennung von Rechteausweitungen im aktuellen Design stark von den spezifizierten Schwellenwerten für die jeweiligen Markierungen ab. Diese Werte wurden für die Arbeit manuell ermittelt. Es ist vorstellbar, dass es auch noch weitere legitime Kommunikationsmuster gibt, als das was in Szenario B dargestellt wurde. Für diese könnten die gewählten Schwellenwerte ein Problem darstellen.

Unabhängig von der Erkennung von der eigentlichen Rechteausweitung wurden Hilfsmittel entwickelt, welche der Analyse und Überwachung von Informationsflüssen auf einem Android-System behilflich sind.

Bei der manuellen Ausführung der Evaluationsszenarien waren leichte Perfromanceeinbußen im Vergleich zu einer reinen Android-Installation spürbar. Die Ausführungsgeschwindigkeit wird zum einen durch TaindDroid selbst um 14% verschlechtert [Enck et al., 2010, S. 2] und zum anderen durch die hier entwickelte Erweiterung, welche jede einzelne Interprozesskommunikation überwacht. Eine genaue Analyse der Performanz der Android-Erweiterung steht noch offen.

7. Ausblick

Mit dem hier erarbeiteten Design und seiner Umsetzung im *FlowGraph*-Dienst wurde eine gute Basis für weitere Forschungsansätze im Bereich der Analyse von dynamischen Informationsflüssen auf Android-Systemen geschaffen.

Weiter Forschungsmöglichkeiten umfassen unter Anderem die Analyse von weiteren möglichen Kantenmarkierungen im Systemgraphen oder auch das hinzufügen weiterer Kantengewichte. Vor allem Ansätze die legitime Nutzung und Missbrauch von Schnittstellen unterscheidbarer machen sind von Interesse, da die Schwellenwerte an den Kommunikationskanten für die jeweiligen Markierungen eine manuelle und fragile Stelle im aktuellen Design darstellen.

Ein weiteres interessantes Thema, welches bei der Arbeit an diesem Projekt abzeichnete, sind mögliche Kommunikationsflüsse zwischen Apps. Die Sammlung und Analyse von echten Apps und deren Informationsflussprofilen würde nicht nur der Ermittlung passender Schwellenwerte für das aktuelle Design hilfreich sein, sondern unter Umständen auch weitere relevante Kommunikationsparameter aufdecken, welche über die Bisherigen hinausgehen.

Die hier umgesetzte Implementierung zielt primär auf die Informationsflüsse zwischen Apps ab. Eine Untersuchung auf Integrierbarkeit in andere Sicherheitskonzepte oder die Möglichkeit der Kombination mit existierenden Projekten mit dem Ziel der Einschränkung von Rechteausweitung wäre von Interesse.

Letztendlich wurden in dieser Arbeit mögliche Optionen für Gegenmaßnahmen nach erfolgreicher Erkennung sehr oberflächlich behandelt. In diesem Bereich sind weitaus spezifischere Maßnahmen vorstellbar, welche Daten vor der Kommunikationsausführung filtern oder anderweitig anpassen.

A. CD

A.1. Inhalte

Auf der CD sind folgende Inhalte:

- Diese Arbeit im PDF-Format
- Patches am Android-Quellcode der in der Arbeit entwickelten Erweiterung
- Quellcode der Test-Apps für die verschiedenen Kommunikationswege
- Quellcode der Apps aus den Evaluationsszenarien

A.2. Aufbau der CD

```
/
├── ba
│   └── Bachelorarbeit_T_Markmann.pdf
└── src
    ├── extension ..... Quellcode in Form von Patches
    ├── testapps ..... Quellcode der Test-Apps
    └── scenario_tests .... Quellcode der Apps aus dem Evaluationsszenarien
```

Abkürzungsverzeichnis

API Application Programming Interface

CORBA Common Object Request Broker Architecture

CRUD Create, Retrieve, Update, Delete

DVM Dalvik Virtuelle Maschine

IDE Integrated Development Environment

IDL Interface Definition Language

IP Internet Protokoll

MAC Mandetory Access Control

RMI Remote Method Invocation

RPC Remote Procedure Call

SDK Software Development Kit

URI Uniform Resource Identifier

VM Virtuelle Maschine

Literaturverzeichnis

Glossary of terms. *Machine Learning*, 30:271–274, 1998. ISSN 0885-6125. URL http://dx.doi.org/10.1023/A:1017181826899. 10.1023/A:1017181826899. 21

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, Apr 2011. vi, 2, 4, 16, 17, 22, 24, 31

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb 2012. 17, 18, 25

Graham Cluley. Android malware steals info from one million phone owners. http://nakedsecurity.sophos.com/2010/07/29/android-malware-steals-info-million-phone-owners/, 2010. URL http://nakedsecurity.sophos.com/2010/07/29/android-malware-steals-info-million-phone-owners/. 1

M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D.S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, 2011. 18

W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6. USENIX Association, 2010. URL http://static.usenix.org/events/osdi10/tech/full_papers/Enck.pdf. 19, 20, 23, 26, 27, 28, 51

Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. Technical Report UCB/EECS-2012-26, EECS Department, University of California, Berkeley, Feb 2012. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-26.html. 1, 10

A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011a. 9, 10, 21

A.P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, 2011b. 18, 19, 31

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. 37

Google. Android architecture. URL `https://developer.android.com/about/versions/index.html`. vi, 6

Google. Content provider basics, 2012. URL `https://developer.android.com/guide/topics/providers/content-provider-basics.html`. 34

T. Harada, T. HORIE, and K. TANAKA. Task oriented management obviates your onus on linux. In *Linux Conference*, 2004. 3

Z. Mednieks, L. Dornin, G.B. Meike, and M. Nakamura. *Programming Android*. O'Reilly Media, Inc., 2011. vi, 7, 8, 9, 11, 12

U. Nicola Carlo. Einblick in die dalvik virtual machine. *IMVS Fokus Report*, 3, 2009. 7

Roman Schlegel, Kehuan Zhang, Xiao yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, 2011. 2, 15

Thorsten Schreiber. Android binder. `http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf`, 2011. 3, 11, 12, 30

Mark Ward. Smartphone security put on test. `http://www.bbc.co.uk/news/technology-10912376`, 8 2010. URL `http://www.bbc.co.uk/news/technology-10912376`. 1

Versicherung über Selbständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28. August 2012 Tobias Markmann