Universität Bremen

Diplomarbeit

Analyse und Beschreibung des Binder-Frameworks zur Interprozesskommunikation unter Android als Grundlage für weiterführende Sicherheitsbetrachtungen

Vorgelegt von Kevin Löhmann, Matrikelnummer 1868373

bei der Universität Bremen, Fachbereich 3 - Informatik

Erstgutachter: Dr. Karsten Sohr

Zweitgutachter: Prof. Dr. Michael Lawo

28. Mai 2015

Eigenständigkeitserklärung

Ich versichere hiermit, dass die von mir vorgelegte Diplomarbeit selbstständig und ohne fremde Hilfe verfasst wurde. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Quellen entnommen sind, wurden als solche kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für diese Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Kurzfassung

Das Binder-Framework bildet die Grundlage der Interprozesskommunikation unter Android. Es ist damit ein sicherheitsrelevanter Bestandteil des Android-Betriebssystems. In der bisher verfügbaren Literatur ist das Binder-Framework stark zusammengefasst beschrieben, eine detaillierte Dokumentation, die als Grundlage für weiterführende Untersuchungen verwendet werden kann, ist nicht verfügbar oder nicht veröffentlicht. In dieser Arbeit sind die Hauptfunktionen des Binder-Frameworks detailliert beschrieben. Dies umfasst insbesondere die Transaktionsverarbeitung, die Adressierung von Services, den Verbindungsaufbau zwischen Apps und die Behandlung von aktiven Binder-Objekten sowie die hierfür erforderlichen strukturellen und funktionalen Grundlagen. Zur Orientierung in weiteren Untersuchungen werden die relevanten Bestandteile der Quelltexte des Android-Systems benannt und in Beispielen herangezogen. Gegen Ende der Arbeit werden einzelne Sicherheitsaspekte zusammengefasst, die im Rahmen dieser Arbeit betrachtet werden konnten.

Inhaltsverzeichnis

1. Einleitung

Dieses Kapitel geht einleitend auf die Motivation sowie die Zielsetzung dieser Arbeit ein. Weiterhin werden die verwendeten Vorgehensweisen zur Gewinnung der dargestellten Informationen beschrieben und die wichtigsten Begriffe festgelegt.

1.1. Motivation

Smartphones sind in den großen Industrienationen ein fester Bestandteil des täglichen Lebens vieler Menschen. Mit einem Anteil von 83,1% im dritten Quartal 2014 am weltweiten am Smartphone-Markt ist Android das wichtigste Smartphone-Betriebssystem [Rivera & van der Meulen, 2014].

Bei Smartphones tritt die Telefonie immer weiter in den Hintergrund und macht im Nutzungsprofil oft nur noch einen kleinen Anteil der Gesamtnutzung aus. Im Vordergrund steht die Nutzung von Anwendungen (Apps), wie z.B.:

- Produktivitätsanwendungen (Termine, Aufgaben, Notizen)
- Kommunikation (E-Mail, SMS, Instant Messaging)
- Soziale Netzwerke (Facebook, Twitter, LinkedIn, etc.)
- Medien (Musik, Fotos, Film, Podcasts)
- Lifestyle-Anwendungen (Fitness, Lesen, Hobbys)
- Weitere Anwendungen (z.B. Fernsteuerung, Heimautomatisierung).

Weiterhin finden Smartphones und Tablets breite Anwendung im geschäftlichen Umfeld und in der Industrie. Hier stehen Anwendungen für Zeit- und Leistungserfassung, Maschinensteuerung, geschäftliche Kommunikation und als Kassensystem im Vordergrund.

Viele der genannten Anwendungen verfügen über eine Anbindung an einen Cloud-Service oder ein unternehmensinternes Backend-System, in dem Daten gespeichert und verarbeitet werden. Dieses breite Anwendungsspektrum wird unter anderem durch die folgenden Eigenschaften moderner Smartphones ermöglicht:

- Hochauflösende Kamera
- Integriertes GPS und Kompass
- Konnektivität über mobile Breitbandverbindungen, Bluetooth, Wireless LAN etc.
- Zusätzliche Sensorik (z.B. Beschleunigung, Bewegung, Luftdruck, Fingerabdruck)

Aufgrund ihrer technischen Eigenschaften, der vielfältigen Anwendungsmöglichkeiten und der auf dem Gerät gespeicherten Daten, bilden Smartphones und Tablets ein attraktives Angriffsziel mit vielfältigen Angriffsvektoren. Ein erfolgreicher Angriff auf ein Smartphone ermöglicht den Zugriff auf personenbezogene und weitere sensible Daten des Anwenders. Weiterhin besteht die Möglichkeit zum Abfangen von Kommunikationsverbindungen oder die Überwachung des Anwenders mit der im Smartphone verfügbaren Sensorik. Die genannten Faktoren implizieren, dass der Schutz des Betriebssystems und der auf dem Gerät gespeicherten Daten eine zentrale Anforderung an Smartphone-Betriebssysteme darstellt.

Unter Android wird für die Kommunikation zwischen Prozessen (Interprozesskommunikation) das eigens für Android entwickelte Binder-Framework verwendet. Über dieses Framework werden sensible Daten ausgetauscht. Darüber hinaus basieren sicherheitsrelevante Systemfunktionen auf dem Binder-Framework.

In der vorhandenen Literatur, wie z.B. [Schreiber, 2011] oder [Chin, Porter Felt, Greenwood, & Wagner, 2011], ist das Framework meist in zusammengefasster Form und mit Fokus auf die allgemeine Funktionsweise oder die Anwendung in Apps beschrieben. Verschiedene Publikationen, wie z.B. [Rosa, 2011] oder [Artenstein & Reviso, 2014] befassen sich mit Teilaspekten des Frameworks. Eine detaillierte Dokumentation, die als Grundlage für die Überprüfung von Sicherheitsaspekten des Betriebssystems genutzt werden kann, ist bisher nicht verfügbar.

1.2. Zielsetzung und Abgrenzung

Ziel dieser Arbeit ist, die detaillierte Beschreibung der Struktur und Funktionsweise des Binder-Frameworks um eine Grundlage und Orientierungshilfe für weiterführende Untersuchungen zur Sicherheit des Frameworks bereit zu stellen.

Der Fokus dieser Arbeit liegt auf der Beschreibung der Zusammenhänge zwischen den einzelnen Komponenten im Kontext der Nachrichtenübermittlung sowie den Mechanismen zur Adressierung von Services unter Android.

Das Binder-Framework umfasst Komponenten in allen Schichten des Android-Betriebssystem, angefangen bei Application Programming Interfaces (APIs) zur Verwendung in der App-Entwicklung bis hin zu einem Treibermodul für den Linux-Kernel. Der Schwerpunkt der Betrachtungen in dieser Arbeit liegt auf den Funktionsweisen der Betriebssystem-Komponenten sowie des Kerneltreibers.

Der Aufbau und die Funktionsweise der APIs für die App-Entwicklung, sowie deren Anwendung sind nicht Bestandteil dieser Arbeit. Weiterhin sind die unter Android verwendeten Berechtigungsmodelle auf Systemfunktionen, wie z.B. das Adressbuch, nicht berücksichtigt, da diese nicht im Binder-Framework realisiert sind.

1.3. Vorgehensweisen und Voraussetzungen

Diese Arbeit basiert auf der statischen Analyse der Quelltexte des Android Open Source Projektes sowie dem Android-spezifischen Linux-Kerneltreiber „binder“. Weiterhin wurden Laufzeitanalysen von einfachen Beispielanwendungen durchgeführt um die Funktionen einzelner Anwendungsfälle zu ermitteln

Alle Untersuchungen wurden auf Basis der Android-Version 4.4.4.r1 verwendet. Für die statische Analyse des Sourcecodes wurde die Entwicklungsumgebung „Eclipse“ in Version 4.3 verwendet. Die Laufzeitanalysen der Beispielanwendungen wurden mittels des Debuggers in der Entwicklungsumgebung auf einem virtuellen Android-Device auf Basis des Android-Emulators „Genymotion“ durchgeführt.

Zur Beschreibung der Funktionsweisen werden viele Beispiele aus den analysierten Quelltexten in den Sprachen C, C++ und Java herangezogen. Ein Teil der Funktionen des Frameworks basiert auf Grundfunktionen des Linux-Kernels und der C-Standardbibliotheken. Für das Verständnis dieser Arbeit sind daher folgende Voraussetzungen erforderlich:

- Grundkenntnisse über Linux und Interprozesskommunikation
- Kenntnis über das Android-Betriebssystem
- Kenntnisse der Programmiersprachen C, C++ und Java
- Grundkenntnisse über den Linux-Kernel

1.4. Begriffsdefinitionen

Der Begriff Binder ist mehrdeutig und wird je nach Kontext als Bezeichnung für eine der folgenden Strukturen verwendet:

- Das gesamte Binder-Framework
- Den Kerneltreiber „binder“
- Die Java- oder C++-Klasse `Binder` bzw. `BBinder`
- Eine Instanz der Java- Klasse `Binder` in einer App
- Eine Instanz der Klasse `BinderProxy` in einer App

Um Ungenauigkeiten zu vermeiden, werden daher in dieser Arbeit die in Tabelle 1.1 angegebenen Begriffe verwendet.

Begriff	**Verwendung**
Binder-Framework	Das gesamte Binder-Framework mit allen Komponenten
Service	Eine Instanz der Klasse `BBinder` in C++ bzw. `Binder` in Java
Binder-Proxy	Eine Instanz der Klasse `BinderProxy` in Java bzw. `BpBinder` in C++

Begriff	Verwendung
Kerneltreiber	Das Linux-Kerneltreiber „binder“
Servicemanager	Die konkrete Implementierung des Programms „servicemanager“ welches den Context-Manager unter Android bereitstellt
Context-Manager	Den abstrakten Context-Manager aus Sicht des Frameworks (nicht an eine konkrete Implementierung gebunden)
Client-Prozess[1]	Ein Prozess, der eine Transaktion an einen Service schickt
Serviceprovider	Der Prozess, der in einer Transaktion des Service bereitstellt.
Transaktionsdaten	Metadaten einer Transaktion
Transaktionsinhalte	Daten, die in einer Transaktion zwischen Prozessen ausgetauscht werden.

Tabelle 1.1: Begriffsdefinitionen

[1] Im Binder-Framework sind alle Prozesse sowohl Client als auch Server. Die Bezeichnungen Client- und Serverprozess beziehen sich nur auf den Kontext einer einzelnen Transaktion.

2. Einführung in Android

Dieses Kapitel gibt einen Überblick über die Ursprünge des Betriebssystems Android sowie über dessen interne Architektur. In diesem Zusammenhang werden die für die Interprozesskommunikation relevanten Komponenten, die Sicherheitsmechanismen des Betriebssystems, die Struktur von Apps und die Eigenschaften des Binder-Framework beschrieben.

2.1. Architekturübersicht

In Abbildung 2.1 ist die technische Architektur von Android schematisch dargestellt.

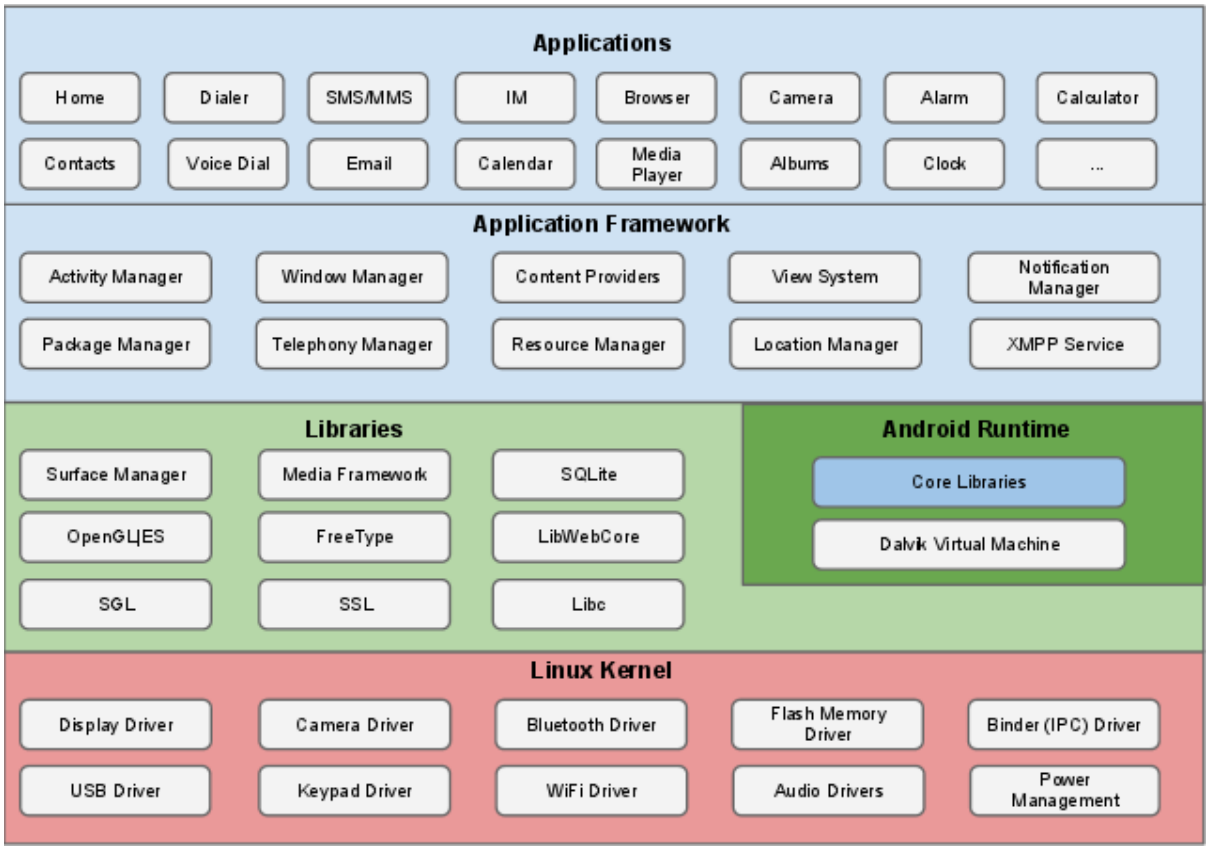

Abbildung 2.1: Architekturübersicht über Android[2]

[2] Quelle: https://source.android.com/devices/tech/security/images/image00.png, Abruf am 15.10.2014

Die technische Basis bilden der Linux-Kernel sowie eine Linux-Laufzeitumgebung. Die Linux-Laufzeitumgebung stellt native Systembibliotheken und Services bereit. Apps werden in der Android-Runtime-Umgebung ausgeführt. Diese basiert auf der Dalvik-VM – einer virtuellen Laufzeitumgebung vergleichbar zur Java-VM. Jede App wird in einer dedizierten Instanz der Dalvik-VM ausgeführt.

Die Android-Runtime-Umgebung stellt das Application Framework bereit. Dies ist eine Sammlung von Bibliotheken welche den Zugriff auf Systemressourcen und -services ermöglichen.

2.2. Start des Betriebssystems

Beim Start des von Android wird eine spezielle Instanz der Dalvik-VM gestartet, in der keine App ausgeführt wird. Diese Instanz wird Zygote genannt. Beim Start einer App wird die Instanz der Dalvik-VM für die App mittels des Systemkommandos `fork()` aus der Zygote erzeugt. Durch dieses Verfahren muss die neue Instanz nicht den vollständigen Initialisierungsprozess durchlaufen. Dadurch wird der Start einer App beschleunigt. Zusätzlich wird durch den Copy-On-Write[3]-Mechanismus von Linux der Speicherverbrauch reduziert.

Direkt nach dem Start der Zygote wird der Android-Systemserver gestartet. Dieser wird unter dem Benutzerkontext des root-Benutzers gestartet und stellt zentrale Services für Apps in der Android-Runtime-Umgebung – die Android Managed Services – bereit.

2.3. App-Komponenten

Für die Entwicklung von Apps werden vom Android Software Development Kit (SDK) vier standardisierte Komponententypen bereitgestellt. In den folgenden Abschnitten werden diese Komponententypen kurz beschrieben. Alle App-Komponenten können über das Binder-Framework als Services aus fremden Apps verwendet werden. Da der Komponen-

[3] siehe Glossar

tentyp für die Funktionsweise des Binder-Frameworks unerheblich ist, wird in dieser Arbeit der Begriff “Service” synonym für alle Komponententypen verwendet.

2.3.1. Activity

Eine Activity stellt eine Bildschirmseite dar. Sie kann aus mehreren Unterkomponenten (Fragments) aufgebaut sein. Die Visualisierung einer Activity wird in einer XML-Definition beschrieben. Die Funktionalität wird in einer korrespondierenden Controller-Klasse in Java implementiert.

2.3.2. Service

Services dienen dazu, Aufgaben im Hintergrund auszuführen oder Funktionalitäten für andere Apps bereitzustellen. Services können unter Android als Started Service oder als Bound Service implementiert werden.

Started Services werden dauerhaft im Hintergrund ausgeführt und können langlaufende Operationen ohne eine Bindung an eine App ausführen (z.B. Abspielen einer Musikdatei). Bound-Services sind an eine andere App-Komponente gebunden und werden beendet, wenn die aufrufende Komponente die Bindung aufhebt oder beendet wird.

2.3.3. Content Provider

Content-Provider ermöglichen den synchronen Zugriff auf Daten fremder Apps. Beispielsweise exportiert die Kontakte-App die Daten des Adressbuchs über einen Content-Provider, so dass Adressdaten aus anderen Apps heraus gelesen werden.

2.3.4. Broadcast Receiver

Broadcast-Receiver ermöglichen es, in einer App auf Ereignisse im System oder anderen Apps zu reagieren. Hierzu werden sie beim System registriert. Ereignisse werden von Apps oder vom System ausgelöst, woraufhin alle für das jeweilige Ereignis registrierten Broadcast-Receiver benachrichtigt werden.

2.4. Sicherheitsmodell

Wie schon in der Einleitung beschrieben, ist Sicherheit eine der wichtigsten Anforderungen an das Betriebssystem Android. Um die Angriffswahrscheinlichkeit auf das Betriebssystem zu reduzieren, werden verschiedene Mechanismen eingesetzt, die in [Android Open Source Project, 2014] umfangreich beschrieben. Im Folgenden werden nur die im Kontext der Interprozesskommunikation relevanten Mechanismen beschrieben.

2.4.1. Linux-Kernel

Android basiert auf dem Linux-Kernel. Da der Kernel Open Source ist und sehr breite Verwendung findet, wird er laufend auf Sicherheit hin geprüft und es werden laufend Sicherheitslücken geschlossen. Entsprechend hat der Linux-Kernel ein hohes Sicherheitsniveau und wird oft in sicherheitssensitiven Bereichen eingesetzt (beispielsweise in Firewall-Betriebssystemen). Der Kernel implementiert verschiedene Sicherheitsmechanismen, wie Speicherschutz und Isolation von Prozessen unterschiedlicher User, so dass Zugriffe auf Ressourcen von Prozessen anderer Benutzer ausgeschlossen sind.

2.4.2. Application Sandboxing

Unter Android wird für jede App bei der Installation ein dedizierter Benutzeraccount im Linux-System angelegt. Die App wird nur unter diesem Benutzer ausgeführt, damit unterliegt sie beim Zugriff auf Daten fremder Apps den Zugriffsbeschränkungen durch den Linux-Kernel. Weiterhin kann eine App anhand der User-ID (UID) des Linux-Benutzers identifiziert werden.

2.4.3. Dateisystemberechtigungen

Unter Android werden die unter Linux gängigen Dateisystemberechtigungen auf User- und Gruppenbasis verwendet. Da jede App eine dedizierte UID hat und die Berechtigungen im Dateisystem restriktiv vergeben werden, ist der Zugriff auf die Dateien fremder Apps nicht möglich. Weiterhin ist die Systempartition, in der das Betriebssystem installiert ist,

schreibgeschützt. Hierdurch werden Manipulationen am Betriebssystem aus Apps heraus verhindert.

2.4.4. SELinux / SEAndroid

Seit Android 4.3 ist das SELinux-Framework Bestandteil des Android-Systems. Dieses Framework ermöglicht die Steuerung von Berechtigungen anhand von Access Control Listen (ACLs) im Linux-Kernel. Für den Einsatz unter Android wurde das Framework um Android-spezifische Aspekte erweitert, daher wird das Framework unter Android als SE-Android bezeichnet.

2.5. Bionic-Bibliothek

Einer der wichtigsten Unterschiede zwischen Android und herkömmlichen Linux-Systemen besteht in der verwendeten C-Library (libc). In herkömmlichen Linux-Systemen für den Server- und Desktop Einsatz wird die GNU C Library (glibc) verwendet. Die glibc ist für den Einsatz unter Android aus zwei Gründen ungeeignet [Brady, 2008]: Zum einen steht die glibc unter der Lesser General Public License (LGPL), die Android-Laufzeitumgebung sollte aber frei von Lizenzen bleiben, die von der General Public License (GPL) abgeleitet sind. Zum anderen ist die glibc sehr umfangreich und führt damit zu einem erhöhten Speicherverbrauch zur Laufzeit. Sie stellt darüber hinaus viele Funktionen bereit, die unter Android nicht benötigt werden oder nicht erwünscht sind. Für Android wurde daher die Bionic-Bibliothek als Ersatz für die glibc entwickelt. Eine detaillierte Beschreibung der Unterschiede zwischen Bionic und glibc liefert [Devos, 2013].

Einer der wichtigsten Unterschiede zwischen glibc und Bionic besteht darin, dass die Bionic-Bibliothek keine Unterstützung für die unter Linux gängigen Mechanismen der Interprozesskommunikation bereitstellt. Unter Linux werden zur Interprozesskommunikation[4] Semaphoren, Shared Memory und Message Queues verwendet. Diese Mechanismen wur-

[4] Die Mechanismen sind in Anhang A.2. beschrieben.

den mit dem System-V von AT&T eingeführt und werden daher als System-V- IPC bezeichnet.

Die Mechanismen der System-V-IPC sind für den Einsatz unter Android nicht geeignet, weil Apps nicht wie herkömmliche Linux-Anwendungen aus der Anwendung heraus beendet werden. Die Beendigung einer App erfolgt unter Android von außen – entweder durch den Benutzer über den Task-Manager oder durch das Betriebssystem wenn wenig Systemressourcen zur Verfügung stehen. Dies bedeutet, dass von der App verwendete Systemressourcen beim Beenden nicht durch die App freigegeben werden können. Dies würde bei Verwendung der Mechanismen der System-V-IPC zu erhöhtem Ressourcenverbrauch und Instabilitäten führen. Der folgende Ausschnitt aus der Dokumentation des Android Source Codes [Android Open Source Project, 2013] illustriert das Problem beim Einsatz der System-V-IPC unter Android:

Android does not support System V IPCs, i.e. the facilities provided by the following standard Posix headers:
<sys/sem.h> / SysV semaphores */*
<sys/shm.h> / SysV shared memory segments */*
<sys/msg.h> / SysV message queues */*
<sys/ipc.h> / General IPC definitions */*
The reason for this is due to the fact that, by design, they lead to global kernel resource leakage.
For example, there is no way to automatically release a SysV semaphore allocated in the kernel when:
- a buggy or malicious process exits
- a non-buggy and non-malicious process crashes or is explicitely killed.
Killing processes automatically to make room for new ones is an important part of Android's application lifecycle implementation. This means that, even assuming only non-buggy and non-malicious code, it is very likely that over time, the kernel global tables used to implement SysV IPCs will fill up.
At that point, strange failures are likely to occur and prevent programs that use them to run properly until the next reboot of the system.
And we can't ignore potential malicious applications. As a proof of concept here is a simple exploit that you can run on a standard Linux box today:...

2.6. Eigenschaften des Binder-Frameworks

Das Binder-Framework basiert auf dem Framework OpenBinder, welches von Be Inc. für das Betriebssystem BeOS entwickelt wurde. Für Android wurde OpenBinder anfangs angepasst und später vollständig neu geschrieben. Eine Alternative zur Entwicklung des Binder-Frameworks wäre die Integration des dbus[5]-Frameworks in Android gewesen. Im Jahr 2009 wurde in der Linux-Community darüber diskutiert, warum dem Binder-Framework der Vorzug gegeben wurde. Die wahrscheinlichste Erklärung ist, dass Diane Hackborn – eine der führenden Entwicklerinnen des Binder-Frameworks – früher bei Be Inc. beschäftigt war und dort das OpenBinder-Framework entwickelt hat [Hellmann, 2013].

Die folgenden Abschnitte geben einen kurzen Überblick über die Eigenschaften des Binder-Frameworks. Die meisten der genannten Funktionen sind in den späteren Kapiteln dieser Arbeit detailliert beschrieben.

2.6.1. Transaktionen

Die Kommunikation über das Binder-Framework erfolgt nachrichtenorientiert in Form von Transaktionen. Transaktionen sind Nachrichten, die synchron im Client-Server-Modell zwischen Prozessen ausgetauscht werden. In der Regel werden durch Transaktionen Methoden an einem Service aufgerufen. Im Rahmen einer Transaktion können beliebige Datentypen übergeben werden. Transaktionen können nur vom Client ausgelöst werden. Der Server kann auf eine Transaktion antworten und hierbei Daten an den Client zurück liefern.

2.6.2. Adressierung

Jeder Service, der über das Binder-Framework angesprochen werden kann, hat eine eindeutige Adresse im Adressraum des Kernels. Diese wird dynamisch zur Laufzeit vergeben. Der Verbindungsaufbau erfolgt unter Einbeziehung von Naming Services.

[5] Siehe Glossar

2.6.3. Sicherheit

Das Binder-Framework arbeitet verbindungsorientiert, bevor ein Client Transaktionen an einen Service schicken kann, muss dieser eine Verbindung zum Service herstellen. Die Verbindungen zwischen Client und Service können nicht von fremden Prozessen genutzt werden.

Bei der Verarbeitung von Transaktionen werden durch den Kerneltreiber die UID und Gruppen-ID (GID) des Absenders in die Transaktion geschrieben. Hierdurch ist es auf Empfängerseite möglich, user- oder gruppenbasierte basierte Berechtigungsmodelle zu implementieren.

Weiterhin bietet das Binder-Framework Unterstützung für die Nutzung des SEAndroid-Frameworks.

2.6.4. Verwaltung von Threadpools

Die Kommunikation über das Binder-Framework erfolgt zwischen einzelnen Threads der beteiligten Prozesse. Um die Kommunikation zwischen Threads zu ermöglichen, verwaltet der Kerneltreiber „binder“ den Threadpool aller Prozesse, die das Binder-Framework nutzen. Hierbei wird zwischen Applikations- und Looper-Threads unterschieden. Looper-Threads sind Empfängerthreads, die Nachrichten empfangen können. Applikations-Threads können Nachrichten senden und nur Antworten empfangen.

2.6.5. Objekt-Mapping

Das Binder-Framework ermöglicht die Übertragung von beliebigen Datentypen in Transaktionen. Bestimmte Objekttypen – sog. aktive Binder-Objekte – werden hierbei gesondert behandelt. Dies ermöglicht den Zugriff auf diese Objekte über Prozessgrenzen hinweg.

2.6.6. Death-Notifications

Das Binder-Framework ermöglicht es, dass Clients benachrichtigt werden, wenn ein angebundener Prozess beendet wird. Hierzu können Clients eine Callback-Methode im Binder-Framework registrieren, die aufgerufen wird, wenn der angebundene Prozess beendet wird.

2.7. Kapitelzusammenfassung

In diesem Kapitel wurde die grundlegende Architektur des Android-Betriebssystems in Bezug auf die Interprozesskommunikation beschrieben.

Die Basis des Android-Betriebssystems bildet der Linux-Kernel sowie eine angepasste Linux-Laufzeitumgebung. Apps werden in der Android-Runtime-Umgebung, die auf der Dalvik-VM basiert, ausgeführt. Für den Zugriff auf Systemdienste steht das Application Framework zur Verfügung.

Beim Start des Betriebssystems wird die Zygote gestartet. Diese dient beim Start von Apps als Ursprungsprozess aus dem die Dalvik-Instanz für die jeweilige App erzeugt wird. Nach der Zygote wird der Android-Systemserver gestartet, der zentrale Dienste für alle Prozesse in der Android-Runtime-Umgebung bereitstellt.

Apps werden aus vier Komponententypen (Activities, Services, Content-Provider, Broadcast-Receiver) aufgebaut. Diese Komponententypen können über das Binder-Framework aus fremden Prozessen heraus genutzt werden.

In Android sind verschiedene Sicherheitsmechanismen implementiert. Hierzu zählen u.a. die durch den Linux-Kernel gegebenen Sicherheitsmechanismen, Application-Sandboxing, restriktive Dateisystemberechtigungen und die Einbindung des SELinux-Frameworks.

Aufgrund der Tatsache, dass Apps unter Android von außen beendet werden, sind die unter Linux sonst üblichen Mechanismen zur Interprozesskommunikation (System-V-IPC) für den Einsatz unter Android nicht stabil einsetzbar. Daher bietet die unter Android verwendete C-Bibliothek Bionic keine Unterstützung für die Mechanismen der System-V-IPC. Stattdessen wird das Binder-Framework zur Interprozesskommunikation verwendet.

Das Binder-Framework ermöglicht den transaktionsorientierten Nachrichtenaustausch im Client-Server-Modell zwischen einzelnen Applikationsthreads. Es unterstützt die Sicherheit in der Interprozesskommunikation durch prozesslokale Verbindungen und die Einbeziehung der Absenderinformationen in Transaktionen. Darüber hinaus ermöglicht es das Objekt-Mapping über Prozessgrenzen hinweg sowie die Versendung von Death-Notifications an Clients.

3. Struktur des Binder-Frameworks

In diesem Kapitel wird der strukturelle Aufbau des Binder-Frameworks beschrieben. Dies beinhaltet die Beschreibung der einzelnen Komponenten und deren Beziehungen untereinander. Hierbei werden auch grundlegende Funktionsweisen der einzelnen Komponenten beschrieben, die Funktionsweise im Gesamtkontext wird in den folgenden Kapiteln ausführlich beschrieben.

3.1. Überblick

Die Basis des Binder-Frameworks bildet der Kerneltreiber „binder“. Dieses ermöglicht die Transaktionsübertragung zwischen den Prozessen im System. Die Kommunikation erfolgt wie in Abbildung 3.1 gezeigt zwischen einzelnen Applikationsthreads. Die Kommunikation zwischen Userspace-Prozess und Kerneltreiber erfolgt über das Device-File `/dev/binder`, dieses wird beim Start einer App durch den Prozess geöffnet. Der Zugriff auf den Kerneltreiber erfolgt ausschließlich aus den nativen Klassen `IPCThreadState` und `ProcessState`, welche zur Prozessinfrastruktur von Apps gehören.

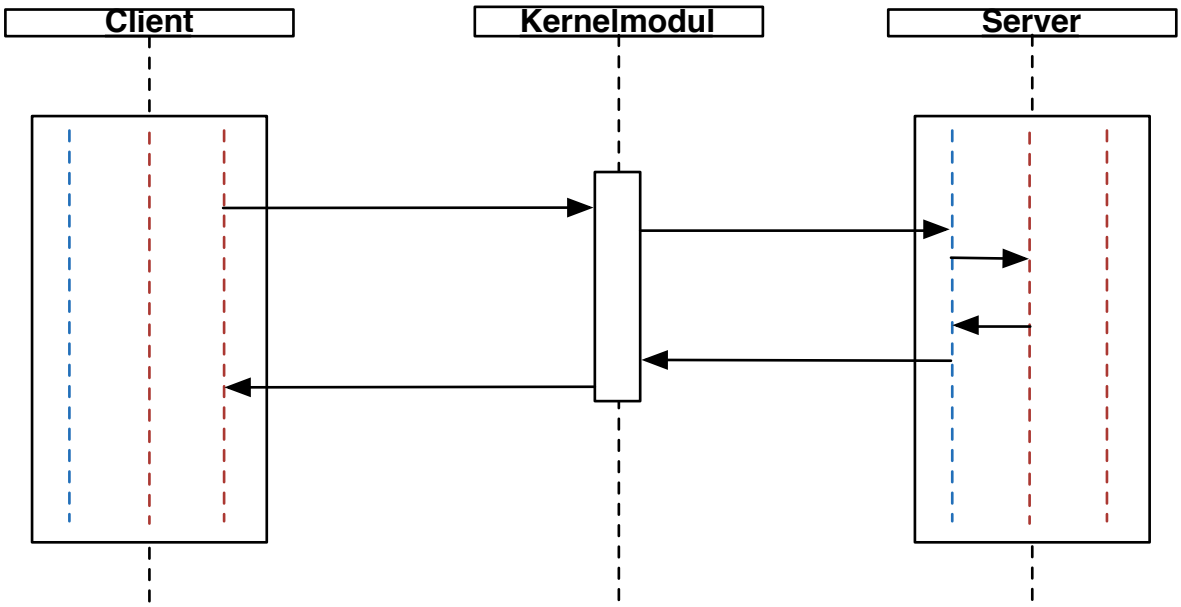

Abbildung 3.1: Darstellung der Kommunikation über das Binder-Framework

In Userspace-Prozessen erfolgt die Nutzung des Frameworks durch das Binder-API. Das Binder-API umfasst Klassen, welche die Nutzung des Frameworks sowohl in Java als auch in C++ ermöglichen. Die Java-Klassen des Binder-API basieren auf den nativen Klassen, die Kopplung erfolgt über das Java Native Interface (JNI). Ein Service wird von Klassen des Typs `Binder` (Java) bzw. `BBinder` (C++) bereitgestellt. Der Zugriff auf den Service im Client erfolgt über Objekte vom Typ `BinderProxy` (Java) bzw. `BpBinder` (C++). Ein Binder-Proxy referenziert genau einen Service. Wie in Abbildung 3.2 gezeigt, kann ein Service von mehren Clients gleichzeitig genutzt werden.

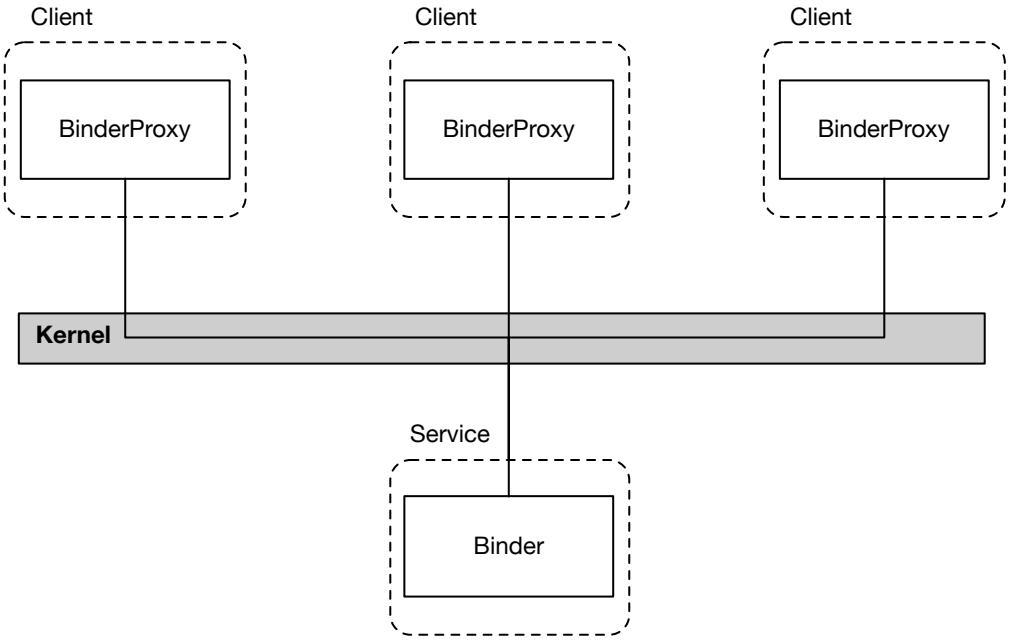

Abbildung 3.2: Beziehungen zwischen Client- und Service-Objekten

Um einen Service nutzen zu können, muss ein Client über einen Naming-Service einen Binder-Proxy anfordern, das den gewünschten Service referenziert. Als Naming-Services wird je nach Anwendungsfall entweder der Servicemanager oder der PackageManager-Service im Android-Systemserver verwendet.

Der Nachrichtenaustausch zwischen Client und Service erfolgt über Transaktionen. Eine Transaktion entspricht einem Methodenaufruf am Service. Die auszuführende Methode wird in der Transaktion durch einen Transaktionscode angegeben. Daten, die in der Trans-

aktion übergeben werden, werden in Containerklassen vom Typ `Parcel` gekapselt. Jede Transaktion enthält die folgenden Parcels:

- `data` für Daten, die vom Client an den Service übergeben werden
- `reply` für Antwortdaten vom Service an den Client.

Transaktionen können bidirektional oder unidirektional (one-way) ausgeführt werden. Bidirektionale Transaktionen erwarten eine Antwort vom Service und werden synchron verarbeitet. Der Clientthread wird hierzu im Kernel so lange blockiert, bis der Service die Antwortdaten bereitgestellt hat. Unidirektionale Transaktionen werden asynchron ausgeführt, das heißt, der Clientthread wird während der Verarbeitung im Service nicht blockiert. Ob eine Transaktion unidirektional oder bidirektional ausgeführt wird, wird über Flags an der Transaktion gesteuert.

```
1  android.os.Parcel _data = android.os.Parcel.obtain();
2  android.os.Parcel _reply = android.os.Parcel.obtain();
3  [ ... ]
4  mRemote.transact(Stub.TRANSACTION_getPid, _data, _reply, 0);
5  [ ... ]
6  _result = _reply.readInt();
```

Listing 3.1: Beispielhafter Aufruf der `transact()`-Methode

Um eine Transaktion auszulösen wird, wie in Listing 3.1 gezeigt, im Client-Prozess die `transact()`-Methode auf dem Binder-Proxy für den jeweiligen Service aufgerufen. Die Methode erwartet die in Tabelle 3.1 angegebenen Argumente. Die Methode `transact()` liefert als Rückgabewert einen Status, der angibt, ob die Transaktion erfolgreich ausgeführt wurde. Die Antwortdaten werden dem Client-Prozess im `reply`-Parcel zur Verfügung gestellt.

Parameter	**Typ**	**Beschreibung**
`Code`	`Int`	Transaktionscode, gibt die auszuführende Aktion an.
`Data`	`Parcel`	Daten, die an den Service übergeben werden
`Reply`	`Parcel`	Daten, die vom Service an den Client übergeben werden

Parameter	Typ	Beschreibung
`Flags`	`Int`	Flags zur Steuerung des Transaktionsverhaltens, bisher ist nur das One-Way-Flag verfügbar.

Tabelle 3.1: Argumente der Methoden `transact()` und `onTransact()`

Bei einer eingehenden Transaktion wird die Methode `onTransact()` im Service ausgeführt. Diese hat die gleichen Argumente wir die `transact()`-Methode des Binder-Proxies. Die Methode `onTransact()` muss den Transaktionscode auswerten und die Ausführung der Aktion an eine Methode im Service delegieren. Ein Beispiel ist in Listing 3.2 angegeben.

```
1  class CustomBinder extends Binder {
2
3      public static final int RETURN_ARGS = 1;
4
5      @Override
6      protected boolean onTransact(int code, Parcel data, Parcel reply, int
           flags) throws RemoteException{
7          switch (code) {
8              case RETURN_ARGS:
9                  return returnArgs(data, reply);
10         }
11     }
12 }
```

Listing 3.2: Beispiel für Methode `onTransact()`

In der App-Entwicklung wird das Binder-API in der Regel nicht direkt genutzt. Der Zugriff erfolgt über das IPC-API welches vom Application Framework bereitgestellt werden. Die Klassen des IPC-API abstrahieren das Framework so weit, dass App-Komponenten (Activities, Services, Broadcast-Receiver, Content-Provider) fremder Prozesse wie lokale Komponenten genutzt werden können. Der Aufruf der `transact()`-Methode ist hierbei in den API-Klassen gekapselt. Die einzelnen Komponenten und die zugehörigen Klassen des Binder-Frameworks sind in Abbildung 3.3 zusammengefasst dargestellt. Diese werden in den folgenden Abschnitten näher beschrieben.

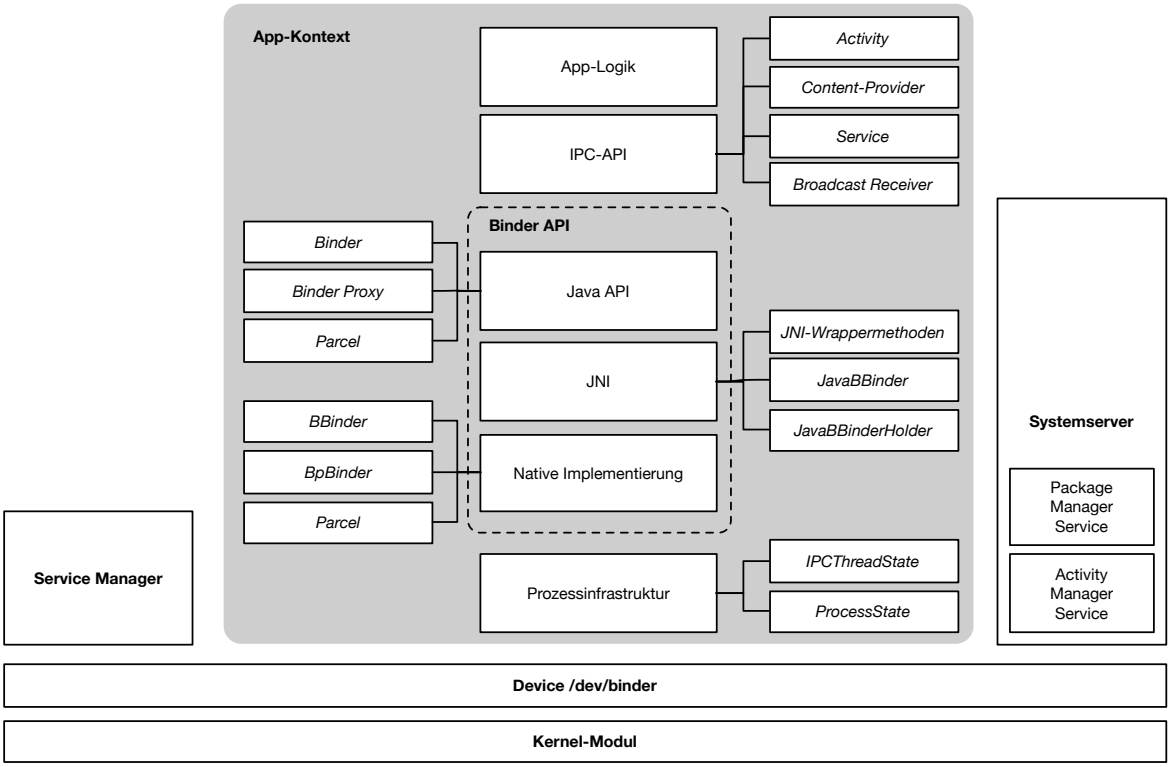

Abbildung 3.3: Komponentenübersicht des Binder-Frameworks

3.2. Binder-API

Strukturell lässt sich die Implementierung des Binder-API in die folgenden Bereiche unterteilen[6]:

- Service (`Binder`) – Implementiert einen Service
- Client (`BinderProxy`) – Repräsentiert einen Service im Client
- Container (`Parcel`) – Container zur Datenübergabe zwischen Client und Service

Die folgenden Abschnitte sind an dieser Struktur orientiert. Das API umfasst für jeden der genannten Bereiche je eine Implementierung in Java und in C++, wobei die Java-Klassen

[6] In Klammern sind die Namen der jeweils relevanten Java-Klassen angegeben. Die Implementierung bezieht verschiedenene Interfaces und abstrakte Klassen, sowie JNI-Wrapper und die nativen Implementierungen der jeweiligen Klassen ein. Der Einfachheit halber werden hier und im Folgenden die Namen der Java-Klassen synonym für die darunter liegende Implementierung verwendet.

auf den nativen Klassen basieren und diese per JNI einbinden. Die folgenden Beschreibungen beziehen sich vorrangig auf Java-Implementierungen, die Prinzipien sind in C++ die gleichen und werden nicht separat beschrieben.

3.2.1. Service (`Binder`)

Das Binder-API für Services umfasst die in Tabelle 3.2 beschriebenen Klassen. Die Beziehungen der Klassen untereinander sind in Abbildung 3.4 dargestellt. Wie in Listing 3.2 gezeigt, erfolgt die Implementierung eines Services in einer Serviceklasse die von der Klasse `Binder` abgeleitet wird. Die Serviceklasse muss die Methode `onTransact()` Implementieren. Diese Methode ist in der Klasse `Binder` ein Stub, der immer `true` zurück liefert.

Klasse	Implementierung	Beschreibung
`Binder`	Java	Stellt für in Java implementierte Apps die Schnittstelle zum Binder-Framework auf Server-Seite dar.
`BBinder`	C++	Native Implementierung der Schnittstelle zum Binder-Framework auf Serverseite
`JavaBBinder`	C++	Generische C++-Implementierung für einen Java-Service.
`JavaBBinderHolder`	C++	Container, der eine Instanz der Klasse Java BBinder hält und für die JNI-Methoden bereitstellt.

Tabelle 3.2: Klassen des Binder-API für Services

Der Aufruf der `onTransact()`-Methode erfolgt indirekt aus der Klasse `IPCThreadState`. Diese Klasse kann nur Methoden an nativen Objekten aufrufen, nicht aber an Java-Objekten. Für Java-basierte Services wird daher zu jeder Instanz der Serviceklasse ein nativer Service vom Typ `JavaBBinder` instanziiert. Dieser Service wird aus dem Java-basierten Service über JNI instanziiert und hat, wie in Listing 3.3 zu sehen, eine Referenz auf den Java-basierten Service. Diese wird in der nativen `onTransact()`-

Implementierung genutzt um die Methode `execTransact()`[7] an der Java-Instanz des Services aufzurufen.

```
1   class JavaBBinder : public BBinder
2   {
3   public:
4       JavaBBinder(JNIEnv* env, jobject object) : mVM(jnienv_to_javavm(env)),
            mObject(env->NewGlobalRef(object))
5       [ ... ]
6       virtual status_t onTransact(uint32_t code, const Parcel& data, Parcel*
            reply, uint32_t flags = 0)
7       {
8           JNIEnv* env = javavm_to_jnienv(mVM);
9           [ ... ]
10          jboolean res = env->CallBooleanMethod(mObject,
                gBinderOffsets.mExecTransact, code, (int32_t)&data,
                (int32_t)reply, flags);
11          [ ... ]
12      }
13      [ ... ]
14  private:
15      JavaVM* const   mVM;
16      jobject const   mObject;
17  };
```

Listing 3.3: Ausschnitt aus der Klasse `JavaBBinder`

Neben dem Aufruf der `execTransact()`-Methode am Java-Service erfolgen auch Methodenaufrufe am `JavaBBinder`-Service aus dem Java-Service heraus. Daher hat der Java-Service eine indirekte Referenz auf den `JavaBBinder`-Service. Die Member-Variable `mObject` des Java-Objektes enthält die Speicheradresse eines natives Objektes vom Typ `JavaBBinderHolder`, welches die Instanz des `JavaBBinder`-Services hält.

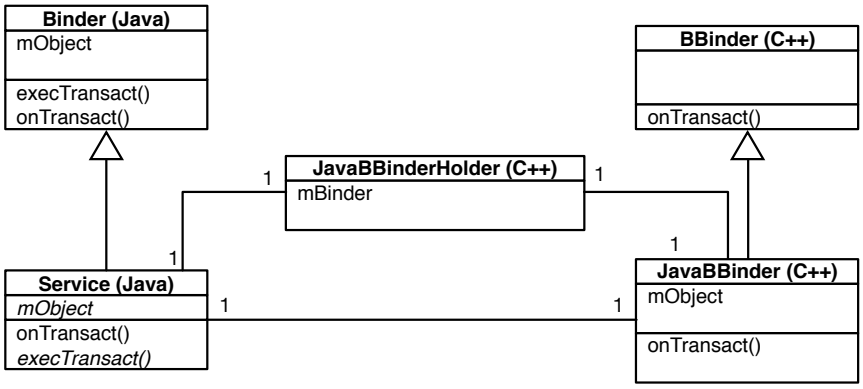

Abbildung 3.4: Klassenbeziehungen bei Java-basierten Services

[7] Die Methode `execTransact()` ruft die Methode `onTransact()` des Java-basierten Services auf, siehe Listing 3.5

3.2.2. Client (`BinderProxy`)

Das Binder-API für Clients umfasst die in Tabelle 3.3 beschriebenen Klassen.

Klasse	Implementierung	Beschreibung
`BinderProxy`	Java	Stellt für in Java implementierte Apps die Schnittstelle zum Binder-Framework auf Client-Seite dar.
`BpBinder`	C++	Native Implementierung des Binder-Proxies, dient als Schnittstelle zum Binder-Framework auf Client-Seite und wird auch über JNI von der Klasse `BinderProxy` genutzt

Tabelle 3.3: Klassen des Binder-API für Clients

Die Klasse `BinderProxy` ist als `final` deklariert, eine Ableitung von dieser Klasse ist daher nicht möglich. Ähnlich wie bei der Klasse `Binder`, wird mit jeder Java-Instanz der Klasse `BinderProxy` eine Instanz der nativen Klasse `BpBinder` instanziiert. Anders als beim Binder wird in der Java-Instanz in der Variablen `mObject` eine direkte Referenz auf das native Objekt gespeichert. Wie in Listing 3.4 gezeigt, ist die `transact()`-Methode in der Java-Implementierung als `native` deklariert. Der Aufruf dieser Methode wird mittels JNI an die `transact()`-Methode des nativen Objektes vom Typ `BpBinder` delegiert.

```
1  final class BinderProxy implements IBinder {
2      [ ... ]
3      public native boolean transact(int code, Parcel data, Parcel reply,
4              int flags) throws RemoteException;
5      [ ... ]
6      private int mObject;
7      private int mOrgue;
8  }
```

Listing 3.4: Auszug aus der Java-Klasse `BinderProxy`

Bei der Instanziierung des nativen Objektes wird dem Konstruktor der Klasse `BpBinder` das Handle für den Proxy übergeben. Das Handle dient zur Adressierung des Services im Kerneltreiber. Die Mechanismen der Adressierung sind in Abschnitt 3.4 zusammengefasst beschrieben.

3.2.3. Container (`Parcel`)

Parcels dienen als Container zur Übertragung von Daten in Transaktionen. Die Klassen des Binder-API für Parcels sind in Tabelle 3.4 beschrieben.

Klasse	Implementierung	Beschreibung
`Parcel`	Java	Dient als Container zur Übertragung von Daten über das Binder-Framework.
`Parcel`	C++	Native Implementierung des Parcels, wird auch über JNI von der Java-Implementierung genutzt

Tabelle 3.4: Klassen des Binder-API für Parcels

Da der Linux-Kernel in C implementiert ist, kann der Kerneltreiber keine komplexen Datentypen wie Java- oder C++-Objekte verarbeiten. Die Daten werden daher beim Hinzufügen zum Parcel für die Verarbeitung im Linux-Kernel vorbereitet. Dieser Prozess wird als „Marshalling“ oder „Flattening“ bezeichnet und ist Teil der Implementierung in der nativen Klasse `Parcel`. Beim Marshalling werden die folgenden primitiven Typen sowie Arrays dieser Typen[8] unterstützt:

- Byte
- Double
- Float
- Int
- Long
- String.

[8] Komplexe Objekttypen müssen auf die primitiven Typen übersetzt werden, siehe hierzu [Hellmann, 2013, Seite 135ff.]

Darüber hinaus werden die folgenden als aktive Binder-Objekte bezeichneten Typen gesondert behandelt[9]:

- Objekte, die von der Klasse `BBinder` ableiten
- Objekte vom Typ `BpBinder`
- Filedeskriptoren

Die Behandlung der aktiven Binder-Objekte ist in Abschnitt 4.5 detailliert beschrieben.

Objekte vom Typ `Binder` und `BinderProxy` werden ausschließlich in Java instanziiert[10], die korrespondierendennativen Objekte werden aus der Java-Anwendung heraus über JNI instanziiert. Bei Objekten vom Typ `Parcel` kann das native Objekt vor dem Java-Objekt instanziiert werden. Dies ist der Fall bei eingehenden Transaktionen im Serviceprovider. Bei dem in Listing 3.3 gezeigten Aufruf der `execTransact()`-Methode in der Klasse `Binder` werden die Speicheradressen der nativen `Parcel`-Objekte an den Java-Service übergeben. Wie in Listing 3.5 zu sehen, erfolgt die Instanziierung der Java-Parcels über die Factory-Methode `obtain()`, hierbei wird die Speicheradresse des zugrundeliegenden nativen Objektes übergeben und in der Variablen `mObject` des Java-Parcels gespeichert.

```
1       // Entry point from android_util_Binder.cpp's onTransact
2       private boolean execTransact(int code, int dataObj, int replyObj,
3               int flags) {
4           Parcel data = Parcel.obtain(dataObj);
5           Parcel reply = Parcel.obtain(replyObj);
6           [ ... ]
7           try {
8               res = onTransact(code, data, reply, flags);
9           } catch (RemoteException e) {
10             [ ... ]
11          }
12          reply.recycle();
13          data.recycle();
14          return res;
15      }
16  }
```

Listing 3.5: Auszug aus der Methode `execTransact()`

[9] Beim Hinzufügen von Java-Objekten der Typen `Binder` oder `BinderProxy` werden tatsächlich die zugrunde liegenden native Objekte verwendet.

[10] Bei nativen Implementierungen entfällt die Instanziierung in Java

Im Client-Prozess erfolgt die Instanziierung der Parcel-Objekte analog zu `Binder` und `BinderProxy` zuerst in der Java-Anwendung.

3.3. Kerneltreiber und Prozessinfrastruktur

Die Kommunikation zwischen Userspace-Prozess und Kerneltreiber erfolgt ausschließlich in den Klassen `IPCThreadState` und `ProcessStat`. Diese bilden die Schnittstelle zwischen den Klassen des Binder-API und dem Kerneltreiber.

In den folgenden Abschnitten sind die Grundlagen der Kommunikation sowie die Funktionen der Klassen der Prozessinfrastruktur beschrieben.

3.3.1. Kommunikation zwischen Prozess und Kerneltreiber

Die Kommunikation zwischen Userspace-Prozess und Kerneltreiber erfolgt über das Device-File `/dev/binder`, das vom Kerneltreiber erzeugt wird. Auf dem Device-File können die File-Operationen `open`, `ioctl`, `mmap`, `poll`, `flush` und `release` ausgeführt werden. Die am häufigsten genutzte Operation ist `ioctl`, da hierüber der Nachrichtenaustausch zwischen Userspace-Prozess und Kernel erfolgt. Die Methoden `poll` und `flush` werden in den Klassen der Prozessinfrastruktur nicht verwendet.

In der Kommunikation zwischen Userspace-Prozess und Kerneltreiber sind die in Tabelle 3.5 angegebenen Nachrichtentypen zu unterscheiden.

Nachrichtentyp	**Beschreibung**
ioctl-Kommando	ioctl-Kommandos werden als Parameter beim Systemcall `ioctl` übergeben und werden direkt in der Methode `binder_ioctl()` des Kerneltreibers ausgewertet.
Binder-Kommando	Binder-Kommandos werden in einen Write-Buffer geschrieben, der über das ioctl-Kommandos `BINDER_WRITE_READ` an den Kerneltreiber übergeben wird.

Nachrichtentyp	Beschreibung
Binder-Response	Binder-Responses werden vom Kerneltreiber in einen Read-Buffer geschrieben, dieser wird im Rahmen des ioctl-Kommandos `BINDER_WRITE_READ` an den Userspace-Prozess übergeben und dort ausgewertet.

Tabelle 3.5: Nachrichtentypen in der Kommunikation mit dem Kerneltreiber

Binder-Kommandos und Binder-Responses sind im Kommunikationsprotokoll des Binder-Kerneltreibers definiert. Die Definition des Binder-Kommunikationsprotokolls ist in Anhang A.5. angegeben.

3.3.1.1. ioctl-Kommandos

Ioctl-Kommandos werden als Argumente zum Systemcall `ioctl()` auf dem Device-File `/dev/binder` an das Kerneltreiber übergeben. Listing 3.6 zeigt beispielhaft den ioctl-Aufruf für das Kommando `BINDER_SET_MAX_THREAD`. Die Verarbeitung im Kerneltreiber erfolgt in der Methode `binder_ioctl()`.

```
1  status_t ProcessState::setThreadPoolMaxThreadCount(size_t maxThreads) {
2      status_t result = NO_ERROR;
3      if (ioctl(mDriverFD, BINDER_SET_MAX_THREADS, &maxThreads) == -1) {
4          result = -errno;
5          ALOGE("Binder ioctl to set max threads failed: %s", strerror(-
   result));
6      }
7      return result;
8  }
```

Listing 3.6: Aufruf des ioctl-Kommandos `SET_MAX_THREADS`

Zusätzlich zum Kommando erwartet der ioctl-Aufruf einen Pointer auf einen Speicherbereich in dem Argumente für das übergebene Kommando liegen. Der erwartete Typ des Arguments variiert je nach verwendetem ioctl-Kommando. In Tabelle 3.6 sind alle vom Kerneltreiber unterstützten ioctl-Kommandos sowie der erwartete Argumenttyp angegeben.

Kommando	Argumenttyp	Funktion
BINDER_WRITE_READ	struct binder_write_read	Übergabe von Binder-Kommandos an den Kerneltreiber und Empfang von Binder-Responses.
BINDER_SET_MAX_THREADS	int	Setzen der maximalen Anzahl Looper-Threads, die der Kerneltreiber beim Userspace-Prozess anfordern kann (siehe Abschnitt 4.2)
BINDER_SET_CONTEXT_MANAGER	-	Registrierung des aktuellen Prozesses als Context-Manager
BINDER_THREAD_EXIT	-	Benachrichtigung über die Beendigung des aktuellen Threads.
BINDER_VERSION	-	Abfrage der Binder-Protokollversion[11]

Tabelle 3.6: Beschreibung der ioctl-Kommandos und ihrer Argumente

Im Rahmen dieser Arbeit werden nur die Ioctl-Kommandos BINDER_WRITE_READ, BINDER_SET_MAX_THREADS und BINDER_SET_CONTEXT_MANAGER betrachtet.

[11] Dieser Arbeit liegt die Version 7 des Binder-Protokolls zugrunde

3.3.1.2. Binder-Kommando `BINDER_WRITE_READ`

Das Kommando `BINDER_WRITE_READ` ist das am häufigsten genutzte ioctl-Kommando. Über dieses Kommando werden die meisten Operationen auf dem Kerneltreiber ausgeführt. Als Parameter muss ein Pointer auf eine Datenstruktur vom Typ `binder_write_read` übergeben werden. Listing 3.7 zeigt, dass die Datenstruktur Referenzen auf einen Write-Buffer und einen Read-Buffer enthält. Im Write-Buffer werden Binder-Kommandos und Argumente für die Kommandos an den Kernel übergeben. Im Read-Buffer werden Binder-Responses und deren Argumente vom Kerneltreiber an den Userspace-Prozess übergeben.

```
1  struct binder_write_read {
2      signed long    write_size;       /* bytes to write */
3      signed long    write_consumed;   /* bytes consumed by driver */
4      unsigned long  write_buffer;
5      signed long    read_size;        /* bytes to read */
6      signed long    read_consumed;    /* bytes consumed by driver */
7      unsigned long  read_buffer;
8  };
```

Listing 3.7: Datenstruktur `binder_write_read`

Wie in Listing 3.8 zu sehen, erfolgt die Verarbeitung des Write-Buffers vor dem Schreiben des Read-Buffers. So ist es möglich, mit einem einzigen ioctl-Aufruf Kommandos an den Kernel zu senden und Antworten zu empfangen.

Der Write-Buffer wird vor der Verarbeitung in den Adressraum des Kernels kopiert. Analog wird der Read-Buffer vor dem Rücksprung in den Userspace-Prozess in dessen Adressraum kopiert.

```
1  case BINDER_WRITE_READ: {
2      struct binder_write_read bwr;
3      [ ... ]
4      if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
5          ret = -EFAULT;
6          goto err;
7       }
8      [ ... ]
9      if (bwr.write_size > 0) {
10         ret = binder_thread_write(proc, thread, (void __user
               *)bwr.write_buffer,
               bwr.write_size, &bwr.write_consumed);
11             [ ... ]
12     }
13     if (bwr.read_size > 0) {
14         ret = binder_thread_read(proc, thread, (void __user
               *)bwr.read_buffer,
               bwr.read_size, &bwr.read_consumed, filp->f_flags & O_NONBLOCK);
15             [ ... ]
16         if (ret < 0) {
17             if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
18                 ret = -EFAULT;
19             goto err;
20     }
21     [ ... ]
22     if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
23        ret = -EFAULT;
24        goto err;
25     }
26     break;
27 }
```

Listing 3.8: Verarbeitung des Kommandos `BINDER_WRITE_READ`

Kommandos und Responses können Argumente erfordern. Diese werden auf das jeweilige Kommando folgend im jeweiligen Buffer gespeichert. Die Struktur sowie die Adressierung der Datenelemente des Write-Buffers ist in Abbildung 3.5 schematisch dargestellt. In der Abbildung ist zu sehen, dass nur der Write-Buffer in den Adressraum des Kernels kopiert wird. Die im Argument zum Kommando BC_TRANSACTION referenzierten Transaktionsdaten bleiben im Adressraum des Userspace-Prozesses. Weiterhin ist zu sehen, dass auch Daten im Adressraum des Kernels referenziert sein können.

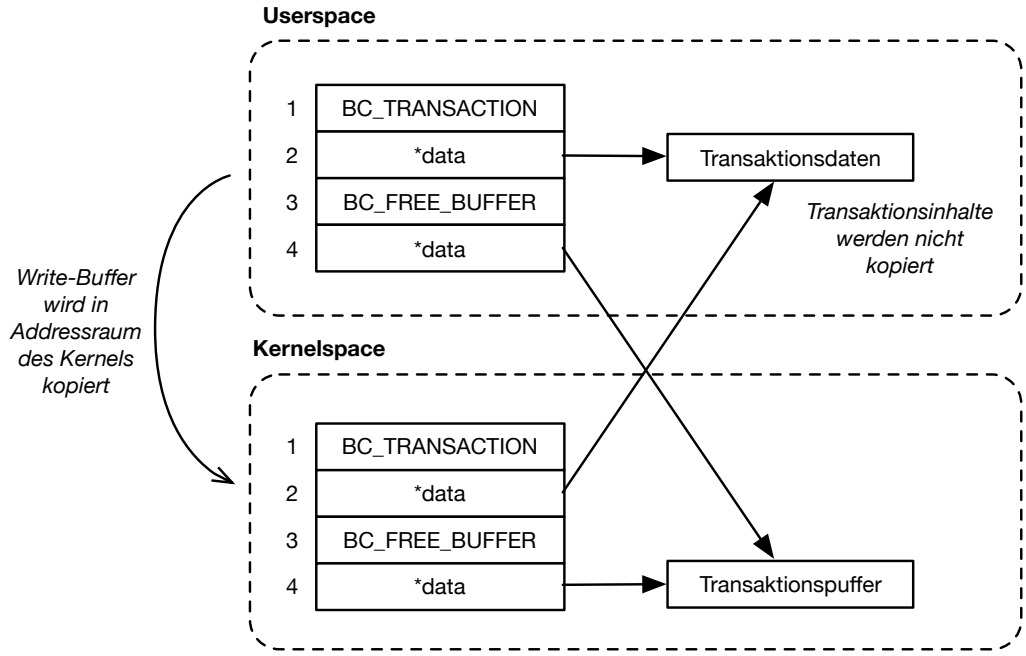

Abbildung 3.5: Struktur des Write-Buffers und Adressierung von Daten

Die Verarbeitung des Write-Buffers wird an die Methode `binder_thread_write()` delegiert. Listing 3.9 zeigt beispielhaft das Auslesen eines Binder-Kommandos und dessen Argumenten aus dem Write-Buffer. Die Ausführung der Binder-Kommandos wird zum Teil an weitere Methoden delegiert.

```
1  int binder_thread_write(struct binder_proc *proc, struct binder_thread
   *thread,
2      void __user *buffer, int size, signed long *consumed)
3  {
4      [ ... ]
5      while (ptr < end && thread->return_error == BR_OK) {
6          if (get_user(cmd, (uint32_t __user *)ptr))
7              return -EFAULT;
8          ptr += sizeof(uint32_t);
9          [ ... ]
10         switch (cmd) {
11             [ ... ]
12             case BC_FREE_BUFFER: {
13                 void __user *data_ptr;
14                 struct binder_buffer *buffer;
15                 if (get_user(data_ptr, (void * __user *)ptr))
16                     return -EFAULT;
17                 ptr += sizeof(void *);
18                 buffer = binder_buffer_lookup(proc, data_ptr);
19                 [ ... ]
20             }
21         }
22     }
23     *consumed = ptr - buffer;
24     [ ... ]
25 }
```

Listing 3.9: Auswertung der Kommandos im Write-Buffer

3.3.1.3. Verarbeitung von Binder-Responses

Binder-Responses werden im Kerneltreiber in der Methode `binder_thread_read()` in den Read-Buffer geschrieben. Dieser wird, wie in Zeile 24 in Listing 3.8 gezeigt, vor dem Rücksprung aus der Methode `binder_ioctl()` in den Adressraum des Userspace-Prozesses kopiert. Die Auswertung der Binder-Responses erfolgt in der Klasse `IPCThreadState`. Binder-Responses können aufgrund des Anwendungsfalls im Userspace-Prozess und der damit verbundenen Behandlung in die folgenden Gruppen untergliedert werden:

- Rückgabewerte für Kommandos
- Nachrichten an den Userspace-Prozess.

Rückgabewerte werden als Antwort auf Binder-Kommandos vom Kernel an den Absenderprozess geschickt um diesen über den Ausführungsstatus des Binder-Kommandos zu benachrichtigen. Kommandos werden in der Methode `wait_for_response()` in der Klasse IPCThreadState an den Kernel geschickt. Der Rückgabewert wird, wie in Listing 3.10 gezeigt, in dieser Methode ausgewertet.

```
1  status_t IPCThreadState::waitForResponse(Parcel *reply, status_t
   *acquireResult)
2  {
3      int32_t cmd;
4      while (1) {
5          if ((err=talkWithDriver()) < NO_ERROR) break;
6          [ ... ]
7          cmd = mIn.readInt32();
8          [ ... ]
9          switch (cmd) {
10         case BR_TRANSACTION_COMPLETE:
11             [ ... ]
12         case BR_DEAD_REPLY:
13             [ ... ]
14         case BR_FAILED_REPLY:
15             [ ... ]
16         case BR_ACQUIRE_RESULT:
17             [ ... ]
18         case BR_REPLY:
19             [ ... ]
20         default:
21             err = executeCommand(cmd);
22             [ ... ]
23         }
24     }
25     [ ... ]
26 }
```

Listing 3.10: Methode `waitForResponse()`

Eingehende Nachrichten an den Userspace-Prozess werden, wie in Listing 3.11 zu sehen, in der Methode `executeCommand()` der Klasse `IPCThreadState` behandelt. Hier erfolgt z.B. der Aufruf der `onTransact()`-Methode am Service bei einer eingehenden Transaktion.

```
1  status_t IPCThreadState::executeCommand(int32_t cmd)
2  {
3      BBinder* obj;
4      RefBase::weakref_type* refs;
5      status_t result = NO_ERROR;
6      switch (cmd) {
7      case BR_ERROR:
8          [ ... ]
9      case BR_OK:
10         break;
11     case BR_ACQUIRE:
12         [ ... ]
13     case BR_RELEASE:
14         [ ... ]
15     case BR_INCREFS:
16         [ ... ]
17     case BR_DECREFS:
18         [ ... ]
19     case BR_ATTEMPT_ACQUIRE:
20         [ ... ]
21     case BR_TRANSACTION:
22         [ ... ]
23     case BR_DEAD_BINDER:
24         [ ... ]
25     case BR_CLEAR_DEATH_NOTIFICATION_DONE:
26         [ ... ]
27     case BR_FINISHED:
28         [ ... ]
29     case BR_NOOP:
30         break;
31     case BR_SPAWN_LOOPER:
32         [ ... ]
33     default:
34         [ ... ]
35     }
36
37    [ ... ]
38 }
```

Listing 3.11: Auszug aus der Methode `executeCommand()`

3.3.2. Öffnen des Binder-Devices

Als einer der ersten Schritte nach dem „forken“ aus der Zygote beim Start einer App, wird ein Objekt vom Typ `ProcessState` instanziiert. Listing 3.13 zeigt den Konstruktor der Klasse. Hier ist zu sehen, dass die Methode `open_driver()` aufgerufen wird.

```
1   static int open_driver()
2   {
3       int fd = open("/dev/binder", O_RDWR);
4       if (fd >= 0) {
5           fcntl(fd, F_SETFD, FD_CLOEXEC);
6           int vers;
7           status_t result = ioctl(fd, BINDER_VERSION, &vers);
8           [ ... ]
9           size_t maxThreads = 15;
10          result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
11          [ ... ]
12      return fd;
13  }
```

Listing 3.12: Öffnen des Kerneldevices

Listing 3.12 zeigt, dass in dieser Methode das Device-File `/dev/binder` geöffnet wird und das ioctl-Kommando `SET_MAX_THREADS` an den Kernel geschickt wird.

```
1   Im Konstruktor der Klasse wird weiterhin die Mapped Memory Region
    initialisiert. Hierdurch wird ein Teil des Kernelspeichers in den Adressraum
    des Userspace-Prozesses gemappt. Dieser Speicherbereich wird zur Übergabe von
    Transaktionsdaten aus dem Kernel an den Userspace-Prozess verwendet. Das
    Memory-Mapping ist in Abschnitt 4.3 näher beschrieben.
    ProcessState::ProcessState()
2       : mDriverFD(open_driver())
3       , mVMStart(MAP_FAILED)
4       , mManagesContexts(false)
5       , mBinderContextCheckFunc(NULL)
6       , mBinderContextUserData(NULL)
7       , mThreadPoolStarted(false)
8       , mThreadPoolSeq(1)
9   {
10      if (mDriverFD >= 0) {
11          mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE |
    MAP_NORESERVE, mDriverFD, 0);
12          [ ... ]
13      }
14      [ ... ]
15  }
```

Listing 3.13: Konstruktor der Klasse `ProcessState`

3.3.3. Datenstrukturen im Kernel

Im Kerneltreiber werden die folgenden Komponenten der Userspace-Prozesse durch interne Datenstrukturen abstrahiert[12]:

- Prozess (`struct binder_proc`)
- Thread (`struct binder_thread`)

[12] Die zugehörige Datenstruktur im Kernel ist in Klammern angegeben.

- Service (`struct binder_node`)
- Binder-Proxy (`struct binder_ref`)

Die Abstraktionsbeziehungen sowie die Beziehungen der Datenstrukturen untereinander sind in Abbildung 3.6 dargestellt. Diese Datenstrukturen werden in den folgenden Abschnitten beschrieben. Der Quelltext zu den Datenstrukturen ist in Anhang A.6. angegeben.

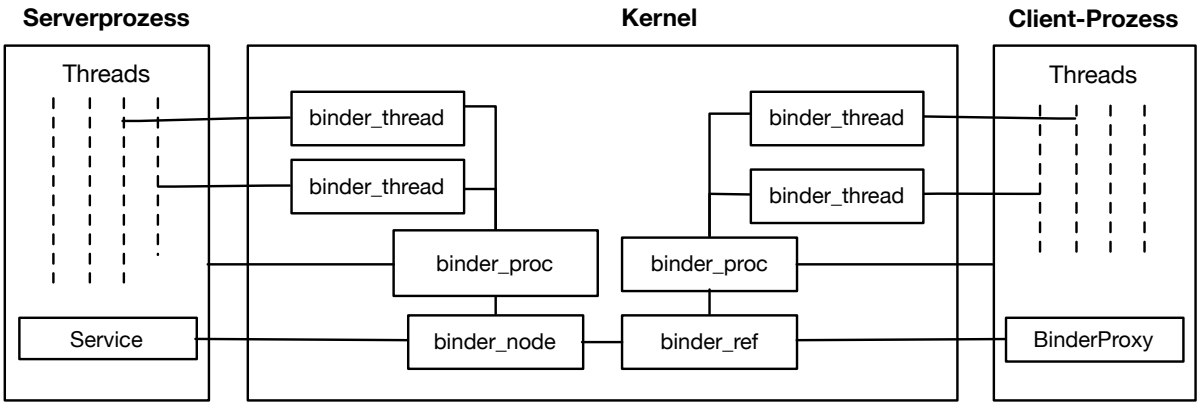

Abbildung 3.6: Beziehungen der Datenstrukturen im Kernel

3.3.3.1. Prozess (`binder_proc`)

Für jeden Prozess wird beim Öffnen des Device-Files `/dev/binder` eine Instanz der Datenstruktur `binder_proc` angelegt und im `private_data`-Bereich[13] des Device-Files gespeichert, so steht die Datenstruktur bei zukünftigen Zugriffen auf das Device-File zur Verfügung. Diese Datenstruktur bildet das Elternelement für alle weiteren prozessbezogenen Datenstrukturen, die im Kernel angelegt werden.

Neben Referenzen zu den anderen Datenstrukturen werden in der `binder_proc`-Struktur Informationen zur Verwaltung des Threadpools, zur Speicherverwaltung und Statistiken zu

[13] Details zur Verwendung des private_data-Bereichs sind in [Corbet, 2005] beschrieben

Kommandos gespeichert. In der `binder_proc`-Struktur wird auch das `task_struct`[14] für den aufrufenden Prozess referenziert.

```
1  static int binder_open(struct inode *nodp, struct file *filp)
2  {
3      struct binder_proc *proc;
4      [ ... ]
5      proc = kzalloc(sizeof(*proc), GFP_KERNEL);
6      if (proc == NULL)
7          return -ENOMEM;
8      get_task_struct(current);
9      proc->tsk = current;
10     INIT_LIST_HEAD(&proc->todo);
11     init_waitqueue_head(&proc->wait);
12     proc->default_priority = task_nice(current);
13     [ ... ]
14     proc->pid = current->group_leader->pid;
15     INIT_LIST_HEAD(&proc->delivered_death);
16     filp->private_data = proc;
17     [ ... ]
18 }
```

Listing 3.14: Initialisierung der `binder_proc`-Struktur

3.3.3.2. Thread (`binder_thread`)

Ein Thread im Userspace-Prozess wird im Kernel durch die Datenstruktur `binder_thread` repräsentiert. Wie in Listing 3.15 gezeigt wird in der Methode `binder_ioctl()` im Kerneltreiber für jeden Thread eine Instanz der Datenstruktur erzeugt und in der `binder_proc`-Struktur des jeweiligen Prozesses referenziert. Die Erzeugung der Struktur erfolgt implizit durch die in Listing 4.5 gezeigte Methode `binder_get_thread()` beim ersten ioctl-Aufruf durch einen Thread. Bei späteren Aufrufen der Methode wird die vormals initialisierte Datenstruktur zurück geliefert.

```
1  static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long
   arg)
2  {
3      int ret;
4      struct binder_proc *proc = filp->private_data;
5      struct binder_thread *thread;
6      [ ... ]
7      thread = binder_get_thread(proc);
8      [ ... ]
```

Listing 3.15: Abrufen der Thread- und Prozessinformationen

[14] Siehe [Rusling, 1999]

Die Datenstruktur `binder_thread` enthält u.a. Statusflags sowie den Transaktionsstack, der in bidirektionalen Transaktionen verwendet wird.

3.3.3.3. Service (`binder_node`)

Ein Service wird im Kerneltreiber durch die Datenstruktur `binder_node` repräsentiert. Zwischen dem Service im Serviceprovider und der zugehörigen `binder_node`-Struktur im Kernel besteht eine 1:1-Beziehung.

Die `binder_node`-Instanzen zu einem Prozess werden in der `binder_proc`-Struktur referenziert. `binder_node`-Strukturen werden nur dann im Adressraum des Kernels angelegt wenn ein Client eine Verbindung zum jeweiligen Service aufbaut.

In der Variablen `cookie` der `binder_node`-Struktur wird die Speicheradresse des nativen `BBinder`-Objektes im Adressraum des Userspace-Prozesses gespeichert. Dies wird, wie in Listing 3.16 zu sehen, beim Empfang von Transaktionen in der Methode `executeCommand()` der Klasse `IPCThreadState` genutzt, um das Objekt für die Ausführung der Transaktion zu bestimmen.

```
1  status_t IPCThreadState::executeCommand(int32_t cmd)
2  {
3      BBinder* obj;
4      RefBase::weakref_type* refs;
5      status_t result = NO_ERROR;
6
7      switch (cmd) {
8      [ ... ]
9      case BR_TRANSACTION:
10         {
11             binder_transaction_data tr;
12             [ ... ]
13             if (tr.target.ptr) {
14                 sp<BBinder> b((BBinder*)tr.cookie);
15                 const status_t error = b->transact(tr.code, buffer, &reply,
                       tr.flags);
16                 if (error < NO_ERROR) reply.setError(error);
17
18             } else {
19                 const status_t error = the_context_object->transact(tr.code,
                       buffer, &reply, tr.flags);
20                 if (error < NO_ERROR) reply.setError(error);
21             }
22             [ ... ]
23             break;
24         [ ... ]
25         }
26     [ ... ]
27 }
```

Listing 3.16: Verwendung des Cookies

3.3.3.4. Binder-Proxy (`binder_ref`)

Ein Binder-Proxy im Userspace-Prozess wird im Kernel durch eine Instanz der Datenstruktur `binder_ref` repräsentiert. Zwischen `BinderProxy` und `binder_ref` besteht eine 1:1 Beziehung.

In der Variablen `node` der `binder_ref`-Struktur wird die Speicheradresse der `binder_node`-Struktur des durch den Proxy repräsentierten Services gespeichert. Die Datenstruktur `binder_ref` bildet damit das Kernelement der Adressierung von Services in der Transaktionsverarbeitung. Alle `binder_ref`-Strukturen zu einem Prozess werden in der `binder_proc`-Struktur in den Listen `refs_by_desc` und `refs_by_node` referenziert.

Die Liste `refs_by_desc` ordnet jeder `binder_ref`-Struktur eine fortlaufende Nummer zu. Diese wird an den Userspace-Prozess übergeben und im `BpBinder`-Objekt als Handle verwendet. Über das Handle wird beim Versenden einer Transaktion die zugehörige `binder_ref`-Struktur identifiziert. Die Nummerierung der `binder_ref`-Strukturen erfolgt prozesslokal. Ein Handle ist damit in anderen Prozessen nicht nutzbar.

3.4. Kapitelzusammenfassung

In diesem Kapitel wurden die wichtigsten Klassen und Datenstrukturen des Binder-Frameworks sowie deren Beziehungen untereinander beschrieben.

Im Userspace-Prozess erfolgt die Nutzung des Binder-Frameworks über die Klassen des Binder-API. Diese sind strukturell in die drei Funktionsblöcke Service (`Binder`), Client (`BinderProxy`) und Datencontainer (`Parcel`) untergliedert. Für jede der Klassen existieren Implementierungen in Java und in C++, wobei die Java-Implementierungen auf den nativen Implementierungen basieren und über JNI eng an diese gekoppelt sind.

Die Kommunikation zwischen Userspace-Prozess und Kerneltreiber erfolgt über die Klasse `IPCThreadState`. Diese abstrahiert das Kommunikationsprotokoll des Binders gegenüber den Klassen des Binder-API. Die Kommunikation erfolgt mittels ioctl-Aufrufen auf dem Device-File `/dev/binder`, welches beim Start einer App geöffnet wird. In der

Kommunikation wird zwischen ioctl-Kommandos, Binder-Kommandos und Binder-Responses unterschieden. Binder-Kommandos und Responses werden zusammen mit erforderlichen Argumenten in Buffern über ioctl-Kommando `BINDER_WRITE_READ` zwischen Userspace-Prozess und Kernel ausgetauscht.

Innerhalb des Kernels werden die Strukturen im Userspace-Prozess durch die Datenstrukturen `binder_proc`, `binder_thread`, `binder_node` und `binder_ref` abstrahiert. Die Datenstrukturen `binder_ref` und `binder_node` bilden hierbei die Kernelemente der Adressierung in der Transaktionsverarbeitung. Die Zusammenhänge der Adressierung sind in Abbildung 3.7 zusammengefasst. Hier ist zu sehen, dass das Handle im `BpProxy`-Objekt zur Identifizierung der zugehörigen `binder_ref`-Struktur im Kernel verwendet wird. Die `binder_ref`-Struktur referenziert die zugehörige `binder_node`-Struktur. Über das Cookie der `binder_node`-Struktur ist das native Binder-Objekt im Serviceprovider referenziert.

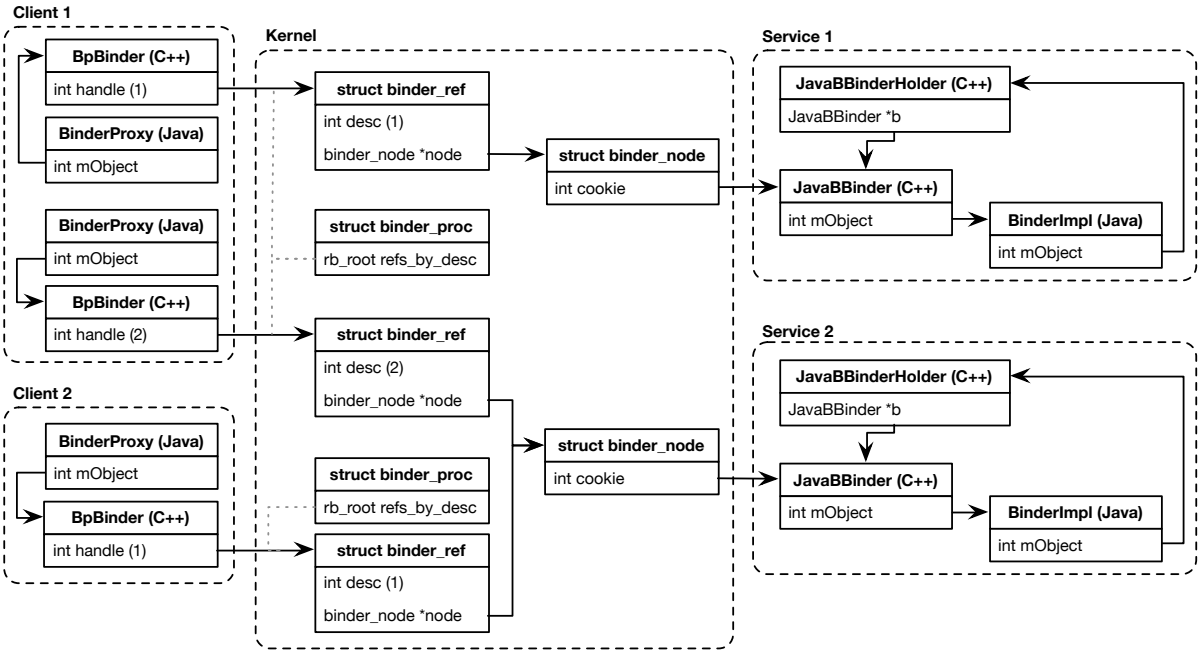

Abbildung 3.7: Adressierung von Objekten im Binder-Framework

4. Funktionen des Binder-Frameworks

Die Kernfunktionalität des Binder-Frameworks ist die Übertragung von Transaktionen zwischen Prozessen. Diese basiert auf weiteren Funktionalitäten des Frameworks. In diesem Kapitel sind die folgenden Funktionen beschrieben, welche die Grundlage für die Transaktionsverarbeitung bilden:

- Nachrichtenverarbeitung im Kerneltreiber
- Verwaltung von Looper-Threads
- Memory Mapping zur Bereitstellung von Transaktionsdaten
- Behandlung aktiver Binder-Objekte
- Namensauflösung für Systemservices

4.1. Nachrichtenverarbeitung im Kernel

Die Kommunikation über das Binder-Framework erfolgt zwischen einzelnen Threads. Hierbei wird zwischen Applikationsthreads und Looper-Threads unterschieden. Applikationsthreads sind Client-Threads, die Nachrichten an den Kerneltreiber schicken können. Looper-Threads sind Empfängerthreads, die auf eingehende Nachrichten warten. Die Verwaltung von Looper-Threads ist in Abschnitt 4.2 beschrieben.

Services sind nicht an bestimmte Looper-Threads gebunden. Looper-Threads können Nachrichten für alle Services des Prozesses empfangen. Dies impliziert, dass der annehmende Thread für eine Transaktion nicht vorab bestimmt werden kann.

Antworten auf eine Nachricht müssen an den Absenderthread der Nachricht geschickt werden. Daher werden Applikationsthreads, die auf Antworten warten, im Kernel blockiert bis die Antwort vorliegt.

In den folgenden Abschnitten werden die Grundprinzipien der Adressierung von Nachrichten und das Blockieren von Threads im Kernel beschrieben.

4.1.1. Work-Items und Todo-Listen

Die Schnittstelle zwischen Kerneltreiber und Userspace-Prozess bilden der Write- und Read-Buffer, die mit dem ioctl-Kommando `BINDER_WRITE_READ` übergeben werden. In der Verarbeitung von Binder-Kommandos werden die Binder-Responses nicht direkt in den Read-Buffer des Empfängerthreads geschrieben. Stattdessen werden im Kernel sog. Work-Items zur Benachrichtigung verwendet. Diese werden an Todo-Listen angehängt die in der Methode `binder_thread_read()` ausgewertet werden. Anhand der Work-Items in der Todo-Liste werden die Binder-Responses in den Read-Buffer des Zielthreads geschrieben. Die verfügbaren Work-Items sind in der in Listing 4.1 gezeigten Datenstruktur `binder_work` definiert.

```
1  struct binder_work {
2  struct list_head entry;
3  enum {
4    BINDER_WORK_TRANSACTION = 1,
5    BINDER_WORK_TRANSACTION_COMPLETE,
6    BINDER_WORK_NODE,
7    BINDER_WORK_DEAD_BINDER,
8    BINDER_WORK_DEAD_BINDER_AND_CLEAR,
9    BINDER_WORK_CLEAR_DEATH_NOTIFICATION,
10 } type;
11 };
```

Listing 4.1: Datenstruktur `binder_work`

Die Todo-Listen sind Member-Variablen der Datenstrukturen `binder_proc` und `binder_thread`. Work-Items, für die der annehmende Thread nicht vorab bestimmt werden kann, wie z.B. Transaktionen an einen Service, werden an die prozessbezogene Todo-Liste. Work-Items, für die der Zielthread bekannt ist, wie z.B. Antworten aus Transaktionen, werden an die threadbezogene Todo-Liste angehängt.

4.1.2. Blockieren von Threads

Threads, die auf Binder-Responses warten, werden im Kerneltreiber in der Methode `binder_thread_read()` blockiert. Dies sind alle Looper-Threads und Applikationsthreads, die auf Antworten warten. Zum Blockieren von Threads werden im Kerneltreiber Wait-

queues[15] verwendet. Die Waitqueues sind Member-Variablen der Datenstrukturen `binder_proc` und `binder_thread`. Analog zu den Todo-Listen werden Looper-Threads über eine prozessbezogene Waitqueue blockiert. Applikationsthreads werden über eine threadbezogene Waitqueue blockiert. Listing 4.2 zeigt die Blockierung von Threads über die prozess- und threadbezogenen Waitqueues.

```
1  static int binder_thread_read(struct binder_proc *proc,
2                struct binder_thread *thread,
3                void  __user *buffer, int size,
4                signed long *consumed, int non_block)
5  {
6      [ ... ]
7      wait_for_proc_work = thread->transaction_stack == NULL &&
8              list_empty(&thread->todo);
9      [ ... ]
10     if (wait_for_proc_work) {
11     [ ... ]
12     if (non_block) {
13        if (!binder_has_proc_work(proc, thread))
14             ret = -EAGAIN;
15     } else
16        ret = wait_event_freezable_exclusive(proc->wait,
               binder_has_proc_work(proc, thread));
17     } else {
18        if (non_block) {
19           if (!binder_has_thread_work(thread))
20             ret = -EAGAIN;
21     } else
22        ret = wait_event_freezable(thread->wait,
               binder_has_thread_work(thread));
23 }
```

Listing 4.2: Blockieren von Threads

Sobald eine Nachricht in der Todo-Liste des Prozesses bzw. des Threads vorliegt, wird der nächste wartende Thread in der korrespondieren Waitqueue durch ein `wake_up`-Signal fortgesetzt. Die Prüfung ob ein Thread fortgesetzt wird, erfolgt in den in Listing 4.3 gezeigten Methoden `binder_has_proc_work()` bzw. `binder_has_thread_work()`. Diese Methoden prüfen zum einen ob Einträge in der Todo-Liste des Prozesses bzw. des Threads vorhanden sind, und ob der Thread angehalten werden darf. Ist das Looper-Flag `BINDER_LOOPER_STATE_NEED_RETURN` gesetzt, dann wird der Thread nicht blockiert.

[15] Siehe [Corbet, 2005]

```
1  static int binder_has_proc_work(struct binder_proc *proc,
2              struct binder_thread *thread)
3  {
4  return !list_empty(&proc->todo) ||
5    (thread->looper & BINDER_LOOPER_STATE_NEED_RETURN);
6  }
7  [ ... ]
8  static int binder_has_thread_work(struct binder_thread *thread)
9  {
10 return !list_empty(&thread->todo) || thread->return_error != BR_OK ||
11   (thread->looper & BINDER_LOOPER_STATE_NEED_RETURN);
12 }
```

Listing 4.3: `binder_has_proc_work()` und `binder_has_thread_work()`

4.2. Verwaltung von Looper-Threads

Der Kerneltreiber verwaltet die Anzahl und den Status der Looper-Threads aller Prozesse. Die Verwaltung der Looper-Threads sowie das Anfordern zusätzlicher Looper-Threads ist in den folgenden Abschnitten beschrieben.

4.2.1. Verwaltung der verfügbaren Looper-Threads

Looper-Threads werden im Userspace-Prozess gestartet und beim Kernel registriert. Um sicherzustellen, dass ein Prozess erreichbar ist, kann der Kerneltreiber zusätzliche Looper-Threads beim Userspace-Prozess anfordern. Wie viele Looper-Threads das Kerneltreiber beim Userspace-Prozess anfordern kann, wird vom Userspace-Prozess durch das ioctl-Kommando `BINDER_SET_MAX_THREADS` festgelegt. Die Behandlung dieses ioctl-Kommandos in der Methode `binder_ioctl()` ist in Listing 4.4 dargestellt. Bei Prozessen, die auf den Klassen der Prozessinfrastruktur basieren, wird die Anzahl der Looper-Threads, welche der Kernel anfordern kann, initial auf 15 festgelegt, diese kann zur Laufzeit geändert werden. Der Userspace-Prozess kann ohne Anforderung vom Kernel beliebig viele Looper-Threads starten.

```
1  binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
2  {
3      [ ... ]
4      struct binder_proc *proc = filp->private_data;
5      [ ... ]
6      case BINDER_SET_MAX_THREADS:
7          if (copy_from_user(&proc->max_threads, ubuf, sizeof(proc-
              >max_threads))) {
8      [ ... ]
9  }
```

Listing 4.4: Setzen der Max-Threads im Kerneltreiber

Im Kerneltreiber werden Informationen zu den verfügbaren Looper-Threads in den in Tabelle 4.1 angegebenen Zählvariablen der Datenstruktur `binder_proc` gespeichert.

Variable	**Bedeutung**
`ready_threads`	Anzahl nicht belegter Looper-Threads, dies umfasst vom Userspace-Prozess gestartete Threads und solche, die vom Kernel angefordert wurden.
`requested_threads`	Anzahl der vom Kernel angeforderten aber noch nicht gestarteten Looper-Threads.
`requested_threads_started`	Anzahl der vom Kernel angeforderten und gestarteten Looper-Threads.
`max_threads`	Maximale Anzahl Looper-Threads, die der Kerneltreiber beim Userspace-Prozess anfordern darf.

Tabelle 4.1: Thread-Counter in der Datenstruktur `binder_proc`

Die Zählvariablen in der `binder_proc`-Struktur werden in den folgenden Fällen manipuliert:

- Vom Userspace-Prozess wird ein neuer Looper-Thread registriert
- Der Kerneltreiber fordert einen neuen Looper-Thread an
- Ein vom Kernel angefordertet Looper-Thread wird registriert
- Ein Looper-Thread nimmt eine Nachricht an
- Ein Thread hat die Bearbeitung der Nachricht beendet und kehrt in den Pool zurück

Looper-Threads müssen beim Kernel registriert werden, damit sie über die Waitqueue des Prozess blockert werden. Hierfür stellt das Binder-Protokoll die in Tabelle 4.2 angebenden Kommandos bereit.

Binder-Kommando	Bedeutung
BC_ENTER_LOOPER	Vom Userspace-Prozess gestarteter Looper-Thread, versetzt den Thread in den Status LOOPER_STATE_ENTERED
BC_REGISTER_LOOPER	Vom Kernel angeforderter Looper-Threads, versetzt den Thread in den Status LOOPER_STATE_REGISTERED
BC_EXIT_LOOPER	Setzt in der binder_thread-Struktur das Statusflag BINDER_LOOPER_STATE_EXITED, womit der Thread nicht mehr als Looper zur Verfügung steht.

Tabelle 4.2: Binder-Kommandos zur Thread-Verwaltung

4.2.2. Verwaltung des Looper-Status

Für jeden Thread wird der Looper-Status in der Variablen `looper` in der Datenstruktur `binder_thread` gespeichert. Der Looper-Status ergibt sich aus den in Tabelle 4.3 angegebenen Flags.

Statusflag	Erklärung
BINDER_LOOPER_STATE_NEED_RETURN	Zeigt an, dass der Looper-Thread in der Methode `binder_thread_read()` nicht blockiert werden darf.
BINDER_LOOPER_STATE_REGISTERED	Der vom Kernel angeforderte Looper-Thread wurde gestartet.
BINDER_LOOPER_STATE_ENTERED	Der applikationsseitig gestartete Thread ist als Looper registriert.
BINDER_LOOPER_STATE_WAITING	Der Looper-Thread ist bereit und wartet auf den Empfang von Nachrichten

Statusflag	Erklärung
BINDER_LOOPER_STATE_INVALID	Bei der Registrierung des Looper-Threads wurde eine ungültige Kommandosequenz ausgeführt
BINDER_LOOPER_STATE_EXITED	Der Looper-Thread hat die Endlosschleife verlassen.

Tabelle 4.3: Statusflags für den Looper-Status

Anhand der Statusflags BINDER_LOOPER_STATE_REGISTERED und BINDER_LOOPER_STATE_ENTERED wird unterschieden, ob der Thread vom Kernel angefordert wurde, oder ob es sich um einen Looper handelt, der vom Userspace-Prozess ohne Anforderung gestartet wurde. Daher schließen sich diese beiden Flags gegenseitig aus. Wird versucht, beide Flags an einem Thread zu setzen, dann wird das Flag BINDER_LOOPER_STATE_INVALID gesetzt und dieser Thread kann nicht als Looper verwendet werden. Wenn an einem Thread das Flag BINDER_LOOPER_NEED_RETURN gesetzt ist, dann wird der Thread in der Methode binder_thread_read() nicht blockiert. Bei der Initialisierung der Datenstruktur binder_thread in der in Listing 4.5 gezeigten Methode binder_get_thread(), wird dieses Flag gesetzt. Dies hat zur Folge, dass bei dem in Abschnitt 6.1 beschriebenen Startprozess einer App zwei Looper-Threads gestartet werden.

```
1  static struct binder_thread *binder_get_thread(struct binder_proc *proc)
2  {
3      struct binder_thread *thread = NULL;
4      struct rb_node *parent = NULL;
5      struct rb_node **p = &proc->threads.rb_node;
6
7      while (*p) {
8        parent = *p;
9        thread = rb_entry(parent, struct binder_thread, rb_node);
10
11       if (current->pid < thread->pid)
12          p = &(*p)->rb_left;
13       else if (current->pid > thread->pid)
14          p = &(*p)->rb_right;
15       else
16          break;
17     }
18     if (*p == NULL) {
19       thread = kzalloc(sizeof(*thread), GFP_KERNEL);
20       if (thread == NULL)
21          return NULL;
22       binder_stats_created(BINDER_STAT_THREAD);
23       thread->proc = proc;
24       thread->pid = current->pid;
25       init_waitqueue_head(&thread->wait);
26       INIT_LIST_HEAD(&thread->todo);
27       rb_link_node(&thread->rb_node, parent, p);
28       rb_insert_color(&thread->rb_node, &proc->threads);
29       thread->looper |= BINDER_LOOPER_STATE_NEED_RETURN;
30       thread->return_error = BR_OK;
31       thread->return_error2 = BR_OK;
32     }
33     return thread;
34 }
```

Listing 4.5: Methode `binder_get_thread()` im Kerneltreiber

4.2.3. Anfordern von Looper-Threads durch den Kernel

Wenn wie in Listing 4.7 gezeigt, keine freien Looper-Threads mehr zur Verfügung stehen, der Threadpool des Userspace-Prozesses aber noch nicht ausgeschöpft ist, dann fordert das Kerneltreiber einen neuen Looper-Thread beim Userspace-Prozess an, indem die Binder-Response `BR_SPAWN_LOOPER` in den Read-Buffer des aktuellen Threads geschrieben wird.

```
1  void ProcessState::spawnPooledThread(bool isMain)
2  {
3      if (mThreadPoolStarted) {
4          String8 name = makeBinderThreadName();
5          ALOGV("Spawning new pooled thread, name=%s\n", name.string());
6          sp<Thread> t = new PoolThread(isMain);
7          t->run(name.string());
8      }
9  }
```

Listing 4.6: Methode `spawnPooledThread()`

Die Binder-Response `BR_SPAWN_LOOPER` wird in der Methode `executeCommand()` der Klasse `IPCThreadState` auswertet, hier wird die in Listing 4.6 gezeigte Methode `spawnPooledThread()` der Klasse `ProcessState` aufgerufen, die einen neuen Looper-Thread erzeugt.

```
1  static int binder_thread_read(struct binder_proc *proc,
2                struct binder_thread *thread,
3                void  __user *buffer, int size,
4                signed long *consumed, int non_block)
5  {
6      [ ... ]
7      thread->looper |= BINDER_LOOPER_STATE_WAITING;
8      if (wait_for_proc_work)
9         proc->ready_threads++;
10     [ ... ]
11     if (wait_for_proc_work)
12        proc->ready_threads--;
13     thread->looper &= ~BINDER_LOOPER_STATE_WAITING;
14     [ ... ]
15     if (proc->requested_threads + proc->ready_threads == 0 &&
16        proc->requested_threads_started < proc->max_threads &&
17        (thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
18        BINDER_LOOPER_STATE_ENTERED)) /* the user-space code fails to */
19        /*spawn a new thread if we leave this out */) {
20           proc->requested_threads++;
21           [ ... ]
22           if (put_user(BR_SPAWN_LOOPER, (uint32_t __user *)buffer))
23              return -EFAULT;
24           binder_stat_br(proc, thread, BR_SPAWN_LOOPER);
25     }
26     return 0;
27 }
```

Listing 4.7: Anforderung eines Looper-Threads

4.3. Memory Mapping und Transaktionspuffer

Inhalte[16], die in einer Transaktion übertragen werden, können im Kerneltreiber aus dem Adressraum des Absenderprozesses gelesen werden. Um die Transaktionsinhalte dem Empfängerprozess zur Verfügung stellen zu können, wird ein Teil des Kernelspeichers in den Adressraum des Empfänger-Prozesses gemappt. Dies erfolgt, wie in Abschnitt 3.3.2 beschrieben, unmittelbar nach dem Öffnen des Device-Files durch den Userspace-Prozess. Der Userspace-Prozess kann auf den gemappten Speicher nur lesend zugreifen.

[16] Transaktionsinhalte sind Daten, die den Parcels hinzugefügt wurden.

Für jeden Prozess wird, wie in Listing 4.8 zu sehen, ein 4MB großer Speicherblock alloziert. Damit ist die Gesamtgröße der Transaktionsinhalte über alle Transaktionen, die gleichzeitig von einem Prozess empfangen werden können, auf 4MB begrenzt. Aufgrund dieser Begrenzung ist das Binder-Framework nicht für die Übertragung großer Datenmengen geeignet – hierfür bietet das Framework die Möglichkeit zur Übergabe von Filedeskriptoren.

Der Pointer auf den gemappten Speicherbereich wird in der Member-Variablen `buffer` der Datenstruktur `binder_proc` gespeichert.

```
1  static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
2  {
3      int ret;
4      struct vm_struct *area;
5      struct binder_proc *proc = filp->private_data;
6      const char *failure_string;
7      struct binder_buffer *buffer;
8      if (proc->tsk != current)
9          return -EINVAL;
10     if ((vma->vm_end - vma->vm_start) > SZ_4M)
11         vma->vm_end = vma->vm_start + SZ_4M;
12     [ ... ]
13     if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
14         ret = -EPERM;
15         failure_string = "bad vm_flags";
16         goto err_bad_arg;
17     }
18     vma->vm_flags = (vma->vm_flags | VM_DONTCOPY) & ~VM_MAYWRITE;
19     [ ... ]
20     proc->buffer = area->addr;
21     [ ... ]
22 }
```

Listing 4.8: Memory Mapping im Kernel

Der gemappte Speicherbereich wird während der Transaktionsverarbeitung in Transaktionspuffer unterteilt. Für jede Transaktion steht ein dedizierter Speicherbereich zur Verfügung, der im Rahmen der Transaktionsverarbeitung alloziert werden muss. Der Puffer wird sowohl in den Transaktionsdaten als auch der `binder_proc`-Struktur referenziert. Nach Abschluss der Transaktion muss der entsprechende Puffer wieder freigegeben werden. Da der Userspace-Prozess keinen Schreibzugriff auf den Transaktionspuffer hat, muss dies durch das Binder-Kommando `BC_FREE_BUFFER` erfolgen.

```
1  static void binder_transaction(struct binder_proc *proc,
2                 struct binder_thread *thread,
3                 struct binder_transaction_data *tr, int reply)
4  {
5      t = kzalloc(sizeof(*t), GFP_KERNEL);
6      [ ... ]
7      t->buffer = binder_alloc_buf(target_proc, tr->data_size,
8      tr->offsets_size, !reply && (t->flags & TF_ONE_WAY));
9      if (t->buffer == NULL) {
10         return_error = BR_FAILED_REPLY;
11         goto err_binder_alloc_buf_failed;
12     }
13     t->buffer->allow_user_free = 0;
14     t->buffer->debug_id = t->debug_id;
15     t->buffer->transaction = t;
16     t->buffer->target_node = target_node;
17 }
```

Listing 4.9: Allozierung eines Transaktionspuffers

4.4. Death-Notifications

Beim Beenden eines Prozesses wird das Device-File `/dev/binder` geschlossen. Hierbei wird für den jeweiligen Prozess die Methode `binder_release()` im Kernel aufgerufen. Diese Methode löscht zum Prozess gehörigen Datenstrukturen (`binder_thread`, `binder_node`, `binder_ref`) und gibt die allozierten Speicherbereiche. Hierdurch werden auch alle `binder_ref`-Strukturen in fremden Prozessen entfernt, welche die `binder_node`-Strukturen des beendeten Prozesses referenzieren.

Wenn eine Nachricht an einen Service geschickt wird, der nicht mehr zur Verfügung steht, dann wird die Nachricht vom Kernel mit der Binder-Response `BR_DEAD_REPLY` beantwortet. Dies betrifft sowohl Binder-Proxies, die eine Transaktion auslösen als auch Services, die eine Antwort an einen Binder-Proxy schicken der während der Verarbeitung im Service beendet wird.

Client-Prozesse können beim Kerneltreiber eine Benachrichtigung über die Beendigung eines Services anfordern. In diesem Fall wird vom Kernel eine Binder-Response vom Typ `BR_DEAD_BINDER` an den jeweiligen Client-Prozess geschickt, wenn der Serviceprovider beendet wird.

Um eine Death Notification anzufordern schickt ein Prozess ein Binder-Kommando vom Typ `BC_REQUEST_DEATH_NOTIFICATION` an den Kernel. Das Kommando erwartet zwei Parameter:

- Das Handle das die `binder_ref`-Struktur identifiziert für die eine Benachrichtigung angefordert wird
- Ein cookie, das die Speicheradresse des Binder-Proxies im Userspace enthält.

Die Information, dass eine Death-Notification angefordert wurde und das Cookie des Binder-Proxies werden in der `binder_ref`-Datenstruktur gespeichert. Beim Entfernen der `binder_ref`-Datenstrukturen in der Verarbeitungskette der `binder_release()`-Methode werden Death-Notifications an alle Binder-Proxies verschickt, die eine Death-Notification angefordert haben. Hierbei wird das vormals gespeicherte Cookie als Parameter zur Binder-Response `BR_DEAD_BINDER` in den Read-Buffer geschrieben.

Die Binder-Response `BR_DEAD_BINDER` wird durch einen Looper-Thread empfangen und in der Klasse `IPCThreadState` in der Methode `executeCommand()` behandelt. Listing 4.10 zeigt, wie anhand des Cookies der Binder-Proxy adressiert wird.

```
1  case BR_DEAD_BINDER:
2  {
3      BpBinder *proxy = (BpBinder*)mIn.readInt32();
4      proxy->sendObituary();
5      mOut.writeInt32(BC_DEAD_BINDER_DONE);
6      mOut.writeInt32((int32_t)proxy);
7  } break;
```

Listing 4.10: Behandlung einer Death Notification

Zur Behandlung der Death-Notifications im Client können am `BinderProxy` mit der Methode `linkToDeath()` sog. DeathRecipients registriert werden. Diese werden in der in Listing 4.10 aufgerufenen Methode `sendOrbituary()` über das Beenden des Services informiert. Die `linkToDeath()`-Methode fordert auch die Death-Notification beim Kerneltreiber an.

4.5. Behandlung aktiver Binder-Objekte

Die nativen Objekttypen `BBinder` und `BpBinder` sowie Dateideskriptoren werden als aktive Binder-Objekte bezeichnet. Diese können in Transaktionen an andere Prozesse übergeben werden. Sie werden hierbei sowohl im Flattening im Parcel als auch in der Ver-

arbeitung im Kernel gesondert behandelt. Anwendungsfälle für die Übertragung von aktiven Binder-Objekten sind z.B.:

- Übergabe Binder-Objekten an Clients bei dem in Abschnitt 6.2 beschriebenen Verbindungsaufbau zu einem Service
- Übergabe eines Filedeskriptors auf Shared-Memory-Bereiche bei Nutzung der Android-Shared Memory Implementierung ashmem[17]
- Übergabe von Filedeskriptoren beim Abspielen von Musikdateien
- Nutzung eines Binder-Objektes als Security-Token[18].

In den folgenden Abschnitten werden die Mechanismen zum Flattening und die Behandlung der aktiven Binder-Objekte im Kernel beschrieben.

4.5.1. Flattening im Parcel

Aktive Binder-Objekte werden über die Methoden `writeStrongBinder()`, `writeWeakBinder()` und `writeFileDescriptor()` zum Parcel hinzugefügt. Hierbei werden sie in die in Listing 4.11 dargestellte Datenstruktur `flat_binder_object` übersetzt, die im Kerneltreiber verarbeitet werden kann. Das Flattening erfolgt in der nativen Implementierung der Klasse Parcel.

[17] Die Shared Memory-Implementierung von Android ist nicht Bestandteil dieser Arbeit.

[18] Binder-Objekte werden z.B in den Services des Systemservers als Security-Tokens verwendet. Diese sind nicht Besatndteil dieser Arbeit. Eine Beschreibung ist unter [Lockwood, 2013] zu finden.

```
1  struct flat_binder_object {
2      unsigned long type;
3      unsigned long flags;
4      union {
5          void *binder;
6          signed long handle;
7      };
8      void *cookie;
9  };
```

Listing 4.11: Datenstruktur `flat_binder_object`

Für jedes aktive Binder-Objekt wird eine Datenstruktur vom Typ `flat_binder_object` erzeugt. Die einzelnen Instanzen der Datenstruktur werden im Array `mObjects` der Klasse `Parcel` referenziert. Je nach Objekttyp wird in der Variablen `type` des `flat_binder_object` einer der in Tabelle 4.4 angegebenen Werte gesetzt.

Objekttyp	**type in flat_binder_object**
`BBinder`	`BINDER_TYPE_BINDER`
`BpBinder`	`BINDER_TYPE_HANDLE`
Filedeskriptor	`BINDER_TYPE_FD`

Tabelle 4.4: Typen aktiver Binder-Objekte im flattening

Das Flattening der Objekttypen `BBinder` und `BpBinder` erfolgt in der in Listing 4.12 gezeigten Methode `flatten_binder()`. Beim Flattening eines `BpBinder`-Objektes wird das Handle des `BpBinder`-Objektes dem `flat_binder_object` hinzugefügt. Beim Flattening eines `BBinder`-Objektes wird die Speicheradresse des `BBinder`-Objektes in der Member-Variablen `cookie` des `flat_binder_object` gespeichert. Das Flattening eines Filedeskriptors erfolgt in der Methode `writeFileDescriptor()`.

```
1  status_t flatten_binder(const sp<ProcessState>& proc,
2      const sp<IBinder>& binder, Parcel* out)
3  {
4      flat_binder_object obj;
5
6      obj.flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
7      if (binder != NULL) {
8          IBinder *local = binder->localBinder();
9          if (!local) {
10             BpBinder *proxy = binder->remoteBinder();
11             if (proxy == NULL) {
12                 ALOGE("null proxy");
13             }
14             const int32_t handle = proxy ? proxy->handle() : 0;
15             obj.type = BINDER_TYPE_HANDLE;
16             obj.handle = handle;
17             obj.cookie = NULL;
18         } else {
19             obj.type = BINDER_TYPE_BINDER;
20             obj.binder = local->getWeakRefs();
21             obj.cookie = local;
22         }
23     } else {
24         obj.type = BINDER_TYPE_BINDER;
25         obj.binder = NULL;
26         obj.cookie = NULL;
27     }
28
29     return finish_flatten_binder(binder, obj, out);
30 }
```

Listing 4.12: Flattening eines Binder-Objektes[19]

4.5.2. Unflattening von aktiven Binder-Objekten

Beim Empfänger einer Transaktion müssen die aktiven Binder-Objekte aus den `flat_binder_object`-Strukturen aufgebaut werden. Dies erfolgt beim Lesen aus dem Parcel in den Methoden `readStrongBinder()` und `readWeakBinder()` für dieObjekttypen `BpBinder` und `BBinder`. Diese nutzen hierfür die in Listing 4.13 gezeigte Methode `unflatten_binder()`. Das unflattening von Filedeskriptoren erfolgt in der Methode `readFileDescriptor()`.

[19] Dies ist die Implementierung des Flattening für einen `StrongBinder`, die Implementierung für einen `WeakBinder` ist etwas umfangreicher. Für das Verständnis der allgemeinen Funktionweise ist diese Implementierung besser geeignet.

```
1  status_t unflatten_binder(const sp<ProcessState>& proc,
2      const Parcel& in, sp<IBinder>* out)
3  {
4      const flat_binder_object* flat = in.readObject(false);
5
6      if (flat) {
7          switch (flat->type) {
8              case BINDER_TYPE_BINDER:
9                  *out = static_cast<IBinder*>(flat->cookie);
10                 return finish_unflatten_binder(NULL, *flat, in);
11             case BINDER_TYPE_HANDLE:
12                 *out = proc->getStrongProxyForHandle(flat->handle);
13                 return finish_unflatten_binder(
14                     static_cast<BpBinder*>(out->get()), *flat, in);
15         }
16     }
17     return BAD_TYPE;
18 }
```

Listing 4.13: Unflattening von `Binder`-Objekten

Beim Unflattening eines `flat_binder_object` mit Typ `BINDER_TYPE_HANDLE` wird ein neues Objekt vom Typ `BpBinder` instanziiert. Hierbei wird das im `flat_binder_object` übergebene Handle verwendet. Wenn im `flat_binder_object` der Typ `BINDER_TYPE_BINDER` gesetzt ist, dann wird die im Member `cookie` gesetzte Speicheradresse als Service interpretiert. Bei einem `flat_binder_object` mit Type `BINDER_TYPE_FD` wird nur das Handle zurückgegeben, dieses identifiziert den im Kernel erzeugten Filedeskriptor.

4.5.3. Behandlung im Kernel

Die aktiven Binder-Objekte werden in der Methode `binder_transaction()`, in der die Transaktionsverarbeitung im Kerneltreiber implementiert ist, gesondert behandelt. In Abschnitt 6.2 wird diese Behandlung im Kontext des Verbindungsaufbaus zu einem Service beschrieben.

4.5.3.1. Behandlung von Binder-Objekten

Die Übergabe eines Objekts vom Typ `BBinder` an einen fremden Prozess erfolgt beim Verbindungsaufbau zum jeweiligen Service. Daher muss beim Empfänger der Transaktion beim Unflattening ein Objekt vom Typ `BpBinder` instanziiert werden. Hierzu wird in der Behandlung im Kernel, wie in Listing 4.14 zu sehen, der Typ des `flat_binder_object` von `BINDER_TYPE_BINDER` in `BINDER_TYPE_HANDLE` geändert.

```
1  case BINDER_TYPE_BINDER:
2  case BINDER_TYPE_WEAK_BINDER: {
3    struct binder_ref *ref;
4    struct binder_node *node = binder_get_node(proc, fp->binder);
5    if (node == NULL) {
6         node = binder_new_node(proc, fp->binder, fp->cookie);
7         [ ... ]
8    }
9    [ ... ]
10   ref = binder_get_ref_for_node(target_proc, node);
11   [ ... ]
12   if (fp->type == BINDER_TYPE_BINDER)
13        fp->type = BINDER_TYPE_HANDLE;
14   else
15        fp->type = BINDER_TYPE_WEAK_HANDLE;
16   fp->handle = ref->desc;
17   [ ... ]
18   } break;
```

Listing 4.14: Behandlung des Typs `BINDER_TYPE_BINDER`

Wenn bei der Übergabe eines `BBinder`-Objektes noch keine `binder_node`-Struktur für den Service im Kernel vorhanden ist, dann wird diese angelegt. Hierbei wird das beim Flattening gespeicherte Cookie in der neu erzeugten `binder_node`-Struktur gespeichert.

Beim Empfängerprozess wird eine `binder_ref`-Struktur erzeugt, sofern nicht schon eine vorhanden ist. Dies erfolgt in der in Listing 4.15 gezeigten Methode `binder_get_ref()`. Die laufende Nummer der `binder_ref`-Struktur wird im flat_`binder_object` als Handle gespeichert.

```
1  static struct binder_ref *binder_get_ref(struct binder_proc *proc,
2                    uint32_t desc)
3  {
4      struct rb_node *n = proc->refs_by_desc.rb_node;
5      struct binder_ref *ref;
6
7      while (n) {
8        ref = rb_entry(n, struct binder_ref, rb_node_desc);
9
10       if (desc < ref->desc)
11          n = n->rb_left;
12       else if (desc > ref->desc)
13          n = n->rb_right;
14       else
15          return ref;
16     }
17     return NULL;
18 }
```

Listing 4.15: Methode `binder_get_ref()` im Kerneltreiber

4.5.3.2. Übergabe eines BinderProxies

Wenn ein `flat_binder_object` vom Typ `BINDER_TYPE_HANDLE` an einen anderen Prozess übergeben wird, dann bedeutet dies, dass auf Absenderseite ein `BpBinder`-Objekt übergeben wurde, und der Empfänger Zugriff auf den entsprechenden Service bekommen soll. Bei diesem Anwendungsfall sind die folgenden Fälle zu unterscheiden:

- Das Objekt wird an einen dritten Prozess übergeben
- Das Objekt wird an den Prozess übergeben der, den Service bereitstellt.

Wie in Listing 4.16 gezeigt, wird im ersten Fall eine `binder_ref`-Struktur erzeugt und der `binder_proc`-Struktur des Empfängerprozesses hinzugefügt. Das Handle der neu erzeugten Struktur wird im `flat_binder_object` an den Empfängerprozess übergeben.

Im zweiten Fall wird der Typ des `flat_binder_object` in `BINDER_TYPE_BINDER` geändert. Hierbei wird das Cookie der entsprechenden `binder_node`-Struktur im `flat_binder_object` gespeichert. Hierdurch wird im Empfängerprozess das vorhandene Service-Objekt referenziert.

```
1  case BINDER_TYPE_HANDLE:
2  case BINDER_TYPE_WEAK_HANDLE: {
3    struct binder_ref *ref = binder_get_ref(proc, fp->handle);
4    [ ... ]
5    if (ref->node->proc == target_proc) {
6         if (fp->type == BINDER_TYPE_HANDLE)
7              fp->type = BINDER_TYPE_BINDER;
8         else
9              fp->type = BINDER_TYPE_WEAK_BINDER;
10        fp->binder = ref->node->ptr;
11        fp->cookie = ref->node->cookie;
12        [ ... ]
13   } else {
14        struct binder_ref *new_ref;
15        new_ref = binder_get_ref_for_node(target_proc, ref->node);
16        [ ... ]
17        fp->handle = new_ref->desc;
18        [ ... ]
19   }
20 } break;
```

Listing 4.16: Behandlung des Typs `BINDER_TYPE_HANDLE`

4.5.3.3. Übergabe eines Dateideskriptors

Wenn ein Dateideskriptor an einen anderen Prozess übergeben wird, dann wird in der Behandlung durch das Kerneltreiber der in der Variablen `handle` des

`flat_binder_object` angegebene Dateideskriptor des Prozesses geladen und auf dieser Basis ein neuer Filedeskriptor im Zielprozess erzeugt. Das Handle für den neuen Filedeskriptor wird in das `flat_binder_object` zurückgeschrieben.

```
1  case BINDER_TYPE_FD: {
2    int target_fd;
3    struct file *file;
4    [ ... ]
5    file = fget(fp->handle);
6    [ ... ]
7    task_fd_install(target_proc, target_fd, file);
8    [ ... ]
9    fp->handle = target_fd;
10 } break;
```

Listing 4.17: Behandlung eines Filedeskriptors im Kernel

4.6. Context-Manager

Da die Adressen der `binder_node`-Strukturen im Kernel zur Laufzeit vergeben werden, müssen diese durch einen Naming-Service aufgelöst werden. Der Naming-Service wird im Binder-Framework vom Context-Manager bereitgestellt. Die Namensauflösung erfolgt mittels Transaktionen, die an den Context-Manager geschickt werden. Da der Context-Manager für alle Prozesse ohne vorherige Namensauflösung erreichbar sein muss, wird die Referenz auf den Context-Manager in der globalen Variablen `binder_context_mgr_node` des Kerneltreibers gespeichert. Alle Transaktionen in denen kein Handle gesetzt ist, werden an den Context-Manager geschickt. Unter Android wird das Programm `servicemanager` beim Start des Betriebssystems als Context-Manager registriert.

Der Context-Manager löst nur die Adressen von Systemservices auf. Die Auflösung der Adressen von App-Komponenten erfolgt im PackageManager-Service des Systemservers.

In den folgenden Abschnitten werden die Registrierung des Context-Managers im Kerneltreiber und die wesentlichen Eigenschaften des Servicemanagers beschrieben.

4.6.1. Registrierung des Context-Managers

Die Registrierung eines Prozesses als Context-Manager erfolgt über das ioctl-Kommando `BINDER_SET_CONTEXT_MGR`. Dieses Kommando wird, wie in Listing 4.18 gezeigt, in der Methode `binder_ioctl()` im Kernel verarbeitet.

```
1   case BINDER_SET_CONTEXT_MGR:
2     if (binder_context_mgr_node != NULL) {
3         pr_err("BINDER_SET_CONTEXT_MGR already set\n");
4         ret = -EBUSY;
5         goto err;
6     }
7     ret = security_binder_set_context_mgr(proc->tsk);
8     if (ret < 0)
9         goto err;
10    if (uid_valid(binder_context_mgr_uid)) {
11        if (!uid_eq(binder_context_mgr_uid, current->cred->euid)) {
12            pr_err("BINDER_SET_CONTEXT_MGR bad uid %d != %d\n",
13                   from_kuid(&init_user_ns, current->cred->euid),
14                   from_kuid(&init_user_ns, binder_context_mgr_uid));
15            ret = -EPERM;
16            goto err;
17        }
18    } else
19        binder_context_mgr_uid = current->cred->euid;
20    binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
21    if (binder_context_mgr_node == NULL) {
22        ret = -ENOMEM;
23        goto err;
24    }
25    binder_context_mgr_node->local_weak_refs++;
26    binder_context_mgr_node->local_strong_refs++;
27    binder_context_mgr_node->has_strong_ref = 1;
28    binder_context_mgr_node->has_weak_ref = 1;
29    break;
```

Listing 4.18: Registrierung des Context-Managers

Bei der Registrierung des Context-Managers wird eine `binder_node`-Struktur angelegt. Anders als bei anderen Services wird im Context-Manager kein Binder-Objekt referenziert. Daher wird bei der Erzeugung der `binder_node`-Struktur kein cookie übergeben. Die erzeugte `binder_node`-Struktur wird in der Variablen `binder_context_manager` des Kerneltreibers gespeichert.

4.6.2. Start des Servicemanagers

Der Servicemanager beim Start des Betriebssystems vor allen anderen Services gestartet. Dies ist erforderlich, da alle Systemservices beim Servicemanager registriert werden müssen. Der Servicemanager ist, wie in Listing 4.19 gezeigt, in der Init-Konfiguration als

`critical` deklariert, das bedeutet, dass der Absturz des Servicemanagers zum Neustart des Betriebssystems führt.

```
1  service servicemanager /system/bin/servicemanager
2      class core
3      user system
4      group system
5      critical
6      onrestart restart healthd
7      onrestart restart zygote
8      onrestart restart media
9      onrestart restart surfaceflinger
10     onrestart restart drm
```

Listing 4.19: Start des Servicemanagers

Nach dem Start registriert sich der Servicemanager beim Kerneltreiber mit dem ioctl-Kommando `BINDER_SET_CONTEXT_MANAGER` als Context-Manager. Dieses Kommando darf nur einmal zur Laufzeit des Kernels ausgeführt werden. Für alle weiteren Aufrufe wird vom Kerneltreiber ein Fehler zurückgegeben. Nach der Registrierung als Context-Manager tritt der Servicemanager in eine Endlosschleife ein und wartet auf eingehende Transaktionen.

4.6.3. Unterstütze Binder-Kommandos

Da der Servicemanager nur einen sehr kleinen Teil des Binder-Protokolls implementiert[20], nutzt er nicht die Klassen der in Abschnitt 3.3 beschriebenen Prozessinfrastruktur. Stattdessen verwendet er eine eigene eingeschränkte Schnittstelle zum Kerneltreiber. Die vom Servicemanager unterstützten Binder-Responses sind in Listing 4.20 gezeigt.

[20] Der Servicemanager benötigt beispielsweise keine Thread-Funktionalität, da er in einem einzigen Thread läuft.

```
1  int binder_parse(struct binder_state *bs, struct binder_io *bio,
2                   uint32_t *ptr, uint32_t size, binder_handler func)
3  {
4    [ ... ]
5          switch(cmd) {
6          case BR_NOOP:
7              break;
8          case BR_TRANSACTION_COMPLETE:
9              break;
10         case BR_INCREFS:
11         case BR_ACQUIRE:
12         case BR_RELEASE:
13         case BR_DECREFS:
14             [ ... ]
15         case BR_TRANSACTION: {
16             [ ... ]
17         case BR_REPLY: {
18             [ ... ]
19         case BR_DEAD_BINDER: {
20             [ ... ]
21         case BR_FAILED_REPLY:
22             [ ... ]
23         case BR_DEAD_REPLY:
24             [ ... ]
25         }
26     }
27 }
```

Listing 4.20: Binder-Responses im Servicemanager

4.6.3.1. Verwaltung von Services

Der Servicemanager stellt die vier in Tabelle 4.5 angegebenen Transaktionscodes zum Hinzufügen und Abfragen von Services bereit. Das Deregistrieren von Services wird nicht unterstützt.

Transaktionscode	**Funktion**
`SVC_MGR_GET_SERVICE` `SVC_MGR_CHECK_SERVICE`	Auflösung des Servicenamens in eine Binder-Adresse. Die Verarbeitung dieser Transaktionscodes ist identisch.
`SVC_MGR_ADD_SERVICE`	Registrierung eines neuen Services beim Servicemanager
`SVC_MGR_LIST_SERVICES`	Auflistung aller registrierten Servicenamen.

Tabelle 4.5: Vom Servicemanager unterstützte Transaktionscodes

Der Servicemanager verwaltet die registrierten Services in einer Liste aus Datenstrukturen von dem in Listing 4.21 gezeigten Typ `svcinfo`. Diese beinhalten u.a. den Namen des Services, einen Pointer auf das Callback-Objekt für die in Abschnitt 4.6.3.2 beschriebenen

Death-Notification und einen `void`-Pointer in dem das Handle für den jeweiligen Service gespeichert wird.

```
1  struct svcinfo
2  {
3      struct svcinfo *next;
4      void *ptr;
5      struct binder_death death;
6      int allow_isolated;
7      unsigned len;
8      uint16_t name[0];
9  };
```

Listing 4.21: Struktur `svcinfo` des Servicemanagers

Zur Registrierung eines Services beim Servicemanager wird das entsprechende Binder-Objekt in einer Transaktion an den Servicemanager übergeben. Aufgrund der Behandlung der aktiven Binder-Objekte erhält der Servicemanager hierdurch ein Handle, welches eine `binder_ref`-Struktur im Kerneltreibe referenziert. Bei der Anfrage eines Services wird dem anfragenden Prozess ein aktives Binder-Objekt vom Typ `BINDER_TYPE_HANDLE` übergeben. Damit wird beim anfragenden Prozess ein Binder-Proxy für den Service erzeugt.

4.6.3.2. Death-Notifications

Bei der Registrierung eines Services wird für den jeweiligen Service eine Death-Notification beim Kernel angefordert. Hierdurch wird im Falle dass ein Service beendet wird, beim Servicemanager die in Listing 4.22 gezeigte Callback-Methode `svcinfo_death()` aufgerufen. Diese setzt das Service-Handle auf 0, damit kann dieser Service nicht mehr aufgelöst werden.

```
1  void svcinfo_death(struct binder_state *bs, void *ptr)
2  {
3      struct svcinfo *si = ptr;
4      ALOGI("service '%s' died\n", str8(si->name));
5      if (si->ptr) {
6          binder_release(bs, si->ptr);
7          si->ptr = 0;
8      }
9  }
```

Listing 4.22: Behandlung der Death Notification im Servicemanager

4.7. Kapitelzusammenfassung

In diesem Kapitel wurden die Funktionen des Binder-Frameworks erläutert, die der Transaktionsverarbeitung zugrunde liegen.

Die Kommunikation über den Kerneltreiber erfolgt zwischen einzelnen Threads. Hierbei wird zwischen Looper-Threads und Applikationsthreads unterschieden. Nachrichten, bei denen der Empfängerthread nicht bestimmbar ist, werden an den Prozess gesendet und durch Looper-Threads verarbeitet. Nachrichten, die an einen bestimmten Thread gerichtet sind, wie z.B. Antworten auf Transkationen, werden direkt an den Zielthread adressiert. Alle Threads, die auf Nachrichten warten, werden im Kernel blockiert.

Um sicherzustellen, dass Prozesse Nachrichten empfangen können, kann der Kerneltreiber zusätzliche Looper-Threads beim Prozess anfordern. Hierzu verwaltet der Kerneltreiber den Threadpool aller Prozesse. Dies umfasst die Anzahl der verfügbaren Looper-Threads und deren Status.

Beim Start einer App wird ein Teil des Kernelspeichers in den Adressraum des Userspace-Prozesses gemappt. In diesem Speicherbereich stellt das Kerneltreiber während der Transaktionsverarbeitung, die Transaktionsinhalte für den Empfängerprozess bereit.

Prozesse können Callback-Methoden beim Kernel registrieren, die ausgelöst werden, wenn ein gebundener Prozess beendet wird. Dies bildet u.a. die Grundlage für die Deregistrierung von Services beim Context-Manager.

Aktive Binder-Objekte, dies sind Objekte vom Typ `BpBinder`, `BBinder` und Filedeskriptoren, werden bei der Transaktionsverarbeitung so behandelt, dass bei der Übergabe eines Services an einen fremden Prozess beim Empfänger ein Binder-Proxy erzeugt wird. Weiterhin ist es möglich Binder-Proxies an dritte Prozesse zu übergeben. Die Behandlung aktiver Binder-Objekte bildet die Grundlage für den Verbindungsaufbau zwischen Client und Service sowie für die Namensauflösung beim Context-Manager.

Der Context-Manager stellt den Naming-Service für Systemservices bereit. Unter Android wird der Servicemanager beim Start des Betriebssystems als Context-Manager registriert.

Services werden registriert, indem der Service in einer Transaktion an den Servicemanager übergeben wird. Bei der Namensauflösung liefert der Servicemanager einen Binder-Proxy an den anfragenden Prozess zurück.

5. Transaktionsverarbeitung

Die Transaktionsverarbeitung ist die Kernfunktion des Binder-Frameworks. Der Kommunikationsfluss in Bezug auf die Komponenten des Frameworks ist in Abbildung 5.1 schematisch dargestellt. Dieser wird in diesem Kapitel detailliert beschrieben, angefangen beim Aufruf der `transact()`-Methode am Binder-Proxy bis zum Aufruf der `onTransact()`-Methode am Service.

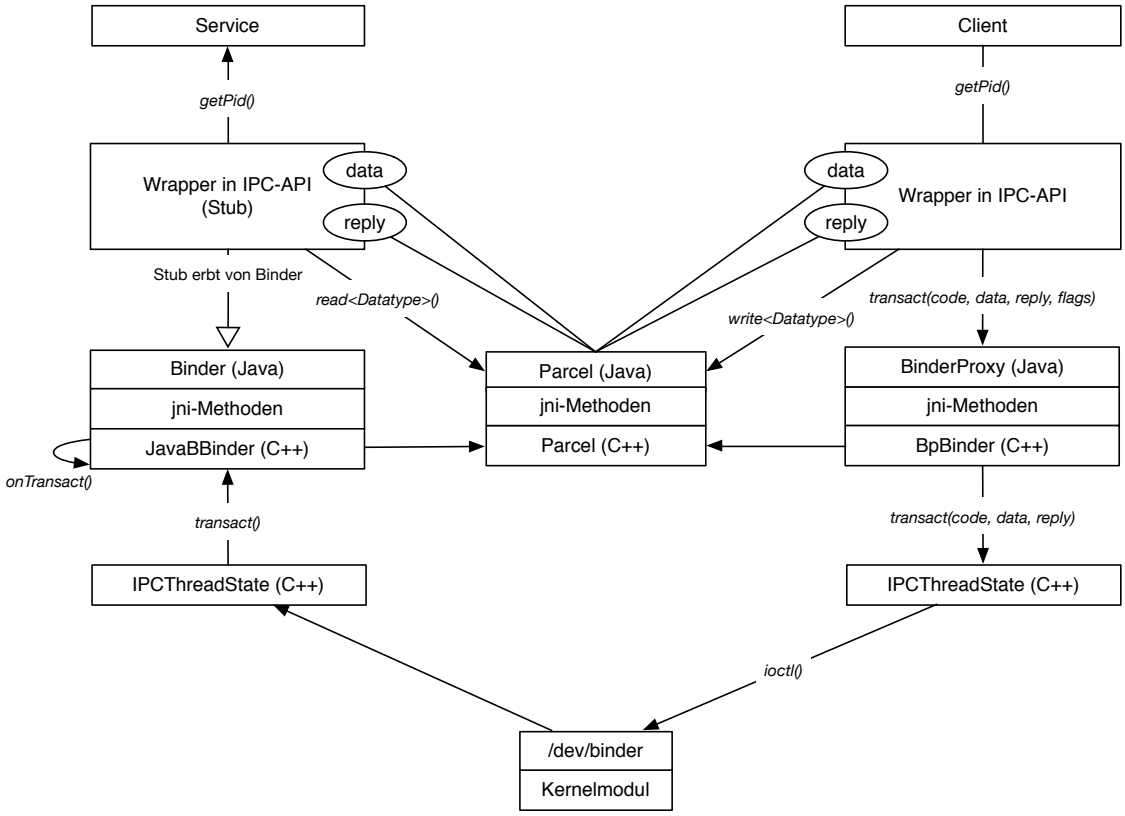

Abbildung 5.1: Kommunikationsfluss in der Transaktionsverarbeitung

5.1. Auslösen der Transaktion im Client-Prozess

Die Transaktion wird durch den Aufruf der `transact()`-Methode am Binder-Proxy ausgelöst. Hierbei werden der Transaktionscode, das `data`-Parcel und das `reply`-Parcel sowie Transaktionsflags als Methodenparameter übergeben. Da das Marshalling der Daten bzw. das Flattening der aktiven Binder-Objekte bereits beim Hinzufügen zum Parcel erfolgt ist, sind hier keine weiteren Schritte notwendig. Die Ausführung der `transact()`-

Methode wird, wie in Listing 5.1 gezeigt, an die threadlokale Instanz der Klasse `IPCThreadState` delegiert. Der Aufruf erfolgt aus der Klasse `BpBinder`, bei diesem Aufruf wird das Handle, welches die `binder_ref`-Struktur im Kerneltreiber identifiziert, übergeben.

```
1  status_t BpBinder::transact(
2      uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
3  {
4      // Once a binder has died, it will never come back to life.
5      if (mAlive) {
6          status_t status = IPCThreadState::self()->transact(
7              mHandle, code, data, reply, flags);
8          if (status == DEAD_OBJECT) mAlive = 0;
9          return status;
10     }
11     return DEAD_OBJECT;
12 }
```

Listing 5.1: Aufruf der `transact()`-Methode in `BpBinder`

Die `transact()`-Methode der Klasse `IPCThreadState` verwendet die in Listing 5.3 gezeigte Methode `writeTransactionData()` um die Transaktionsdaten für den Kernel vorzubereiten. Hierbei wird das Binder-Kommando `BC_TRANSACTION` in den Write-Buffer geschrieben.

```
1  status_t IPCThreadState::transact(int32_t handle,
2                                    uint32_t code, const Parcel& data,
3                                    Parcel* reply, uint32_t flags)
4  {
5      status_t err = data.errorCheck();
6      flags |= TF_ACCEPT_FDS;
7      [ ... ]
8      if (err == NO_ERROR) {
9          LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
10             (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
11         err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data,
   NULL);
12     }
13     [ ... ]
14     if ((flags & TF_ONE_WAY) == 0) {
15         [ ... ]
16         if (reply) {
17             err = waitForResponse(reply);
18         } else {
19             Parcel fakeReply;
20             err = waitForResponse(&fakeReply);
21         }
22         [ ... ]
23     } else {
24         err = waitForResponse(NULL, NULL);
25     }
26     return err;
27 }
```

Listing 5.2: `transact()`-Methode in der Klasse `IPCThreadState`

In der in Listing 5.3 gezeigten Methode `writeTransactionData()` wird eine Datenstruktur vom Typ `binder_transaction_data`, die auch im Kerneltreiber verwendet wird, aufgebaut. Hierbei werden folgende Informationen in die Datenstruktur geschrieben.

- Das Handle aus der Klasse `BpBinder` zur Identifizierung der `binder_ref`-Struktur
- Transaktionscode der Transaktion
- Transaktionsflags
- Pointer auf das Array mit den primitiven Daten im Data-Parcel
- Pointer auf das Array mit den aktiven Binder-Objekten im Parcel

```
1  status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t
   binderFlags,
2      int32_t handle, uint32_t code, const Parcel& data, status_t*
   statusBuffer)
3  {
4      binder_transaction_data tr;
5
6      tr.target.handle = handle;
7      tr.code = code;
8      tr.flags = binderFlags;
9      tr.cookie = 0;
10     tr.sender_pid = 0;
11     tr.sender_euid = 0;
12
13     const status_t err = data.errorCheck();
14     if (err == NO_ERROR) {
15         tr.data_size = data.ipcDataSize();
16         tr.data.ptr.buffer = data.ipcData();
17         tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
18         tr.data.ptr.offsets = data.ipcObjects();
19     } else if (statusBuffer) {
20         [ ... ]
21     } else {
22         return (mLastError = err);
23     }
24     mOut.writeInt32(cmd);
25     mOut.write(&tr, sizeof(tr));
26     return NO_ERROR;
27 }
```

Listing 5.3: Methode `writeTransactionData()`

Nach dem Aufbau der `transaction_data`-Struktur werden das übergebene Binder-Kommando (`BC_TRANSACTION`) und der Pointer auf die `transaction_data`-Struktur in das Parcel `mOut` der Klasse `IPCThreadState` geschrieben. Hierdurch wird in dem Parcel der Write-Buffer, der später im ioctl-Call an den Kernel übergeben wird, aufgebaut. Nach der Vorbereitung der Transaktionsdaten wird durch die `transact()`-Methode die bereits

in Abschnitt 3.3.1.3 beschriebene Methode `waitForResponse()` aufgerufen. Diese Methode delegiert die Kommunikation mit dem Kerneltreiber an die in Listing 5.4 gezeigte Methode `talkWithDriver()`.

In der Methode `talkWithDriver()` wird die Datenstruktur `binder_write_read` initialisiert die den Write- und den Read-Buffer enthält. Der Write-Buffer wird hierbei so initialisiert, dass er auf die Speicheradresse des `data`-Arrays des `mOut`-Parcels zeigt.

```
1  status_t IPCThreadState::talkWithDriver(bool doReceive)
2  {
3      [ ... ]
4      binder_write_read bwr;
5      [ ... ]
6      bwr.write_buffer = (long unsigned int)mOut.data();
7      [ ... ]
8          if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
9              err = NO_ERROR;
10         else
11             err = -errno;
12     [ ... ]
13 }
```

Listing 5.4: Methode `talkWithDriver()`

5.2. Verarbeitung im Kernel

Wie in Abschnitt 3.3.1 beschrieben, erfolgt die Verarbeitung der ioctl-Kommandos und des Write-Read-Buffers in der Methode `binder_ioctl()` im Kerneltreiber, wobei zuerst die Kommandos im Write-Buffer bearbeitet werden und der Read-Buffer nach Abschluss der Verarbeitung gefüllt wird. Die Auswertung des Write-Buffers erfolgt im Kerneltreiber durch die Methode `binder_thread_write()`. Die Verarbeitung für das Kommando `BC_TRANSACTION` wird an die Methode `binder_transaction()` delegiert. Hierzu wird, wie in Listing 5.5 gezeigt, in der Methode `binder_thread_write()` eine neue Datenstruktur vom Typ `binder_transaction_data` erzeugt. Die im Write-Buffer referenzierten Daten im Adressraum des Absenderprozesses werden in den Kernelspeicher kopiert. Anschließend erfolgt der Aufruf der Methode `binder_transaction()`, wobei neben den Transaktionsdaten die Datenstrukturen `binder_proc` und `binder_thread` des Absenderprozesses und Absenderthreads als Parameter übergeben werden.

```
1  int binder_thread_write(struct binder_proc *proc, struct binder_thread
   *thread,
2          void __user *buffer, int size, signed long *consumed)
3  {
4     uint32_t cmd;
5     void __user *ptr = buffer + *consumed;
6     void __user *end = buffer + size;
7
8     switch (cmd) {
9        [ ... ]
10       case BC_TRANSACTION:
11       case BC_REPLY: {
12          struct binder_transaction_data tr;
13          if (copy_from_user(&tr, ptr, sizeof(tr)))
14             return -EFAULT;
15          ptr += sizeof(tr);
16          binder_transaction(proc, thread, &tr, cmd == BC_REPLY);
17          break;
18       [ ... ]
19    }
20    return 0;
21 }
```

Listing 5.5: Aufruf der `binder_transaction()`-Methode

Die Methode `binder_transaction()` verarbeitet sowohl Transaktionen vom Binder-Proxy an den Service als auch Antworten vom Service an den Binder-Proxy. Die folgenden Code-Beispiele beziehen sich nur auf den Fall der Transaktion vom Client an den Service. Daher sind die folgenden Code-Beispiele stark gekürzt und zeigen nur die für diesen Fall relevanten Ausschnitte der `binder_transaction()`-Methode. Im ersten Schritt werden die für die Verarbeitung erforderlichen Datenstrukturen initialisiert (Listing 5.6).

```
1  static void binder_transaction(struct binder_proc *proc,
2                 struct binder_thread *thread,
3                 struct binder_transaction_data *tr, int reply)
4  {
5      struct binder_transaction *t;
6      struct binder_work *tcomplete;
7      size_t *offp, *off_end;
8      struct binder_proc *target_proc;
9      struct binder_thread *target_thread = NULL;
10     struct binder_node *target_node = NULL;
11     struct list_head *target_list;
12     wait_queue_head_t *target_wait;
13     struct binder_transaction *in_reply_to = NULL;
14     struct binder_transaction_log_entry *e;
15     uint32_t return_error;
16     [ ... ]
```

Listing 5.6: Initialisierung der Datenstrukturen in `binder_transaction()`

Anhand des in den Transaktionsdaten gelieferten Handles wird über die Methode `binder_get_ref()` das Ziel der Transaktion bestimmt. Anhand der von der Methode gelieferten `binder_ref`-Struktur sind implizit auch die `binder_node`-Struktur des Services

und die `binder_proc`-Struktur des Zielprozesses bekannt. Damit werden die Todo-Liste und Waitqueue des Prozesses als `target_list` und `target_queue` verwendet. In Listing 5.7 ist zu sehen, dass auch die Todo-Liste und Waitqueue eines Threads verwendet werden können, wenn der Zielthread bestimmt werden kann. Dies ist bei Antworten der Fall.

```
1  [ ... ]
2  if (reply) {
3    [ ... ]
4  } else {
5    if (tr->target.handle) {
6         struct binder_ref *ref;
7         ref = binder_get_ref(proc, tr->target.handle);
8         [ ... ]
9         target_node = ref->node;
10   } else {
11        target_node = binder_context_mgr_node;
12        [ ... ]
13   }
14   [ ... ]
15   target_proc = target_node->proc;
16   [ ... ]
17 }
18 if (target_thread) {
19   e->to_thread = target_thread->pid;
20   target_list = &target_thread->todo;
21   target_wait = &target_thread->wait;
22 } else {
23   target_list = &target_proc->todo;
24   target_wait = &target_proc->wait;
25 }
26 [ ... ]
```

Listing 5.7: Bestimmung der Zielstrukturen für die Transaktion

Die Metadaten einer Transaktion werden in einer Datenstruktur vom Typ `binder_transaction` verwaltet. Dies umfasst u.a. Absender und Ziel der Transaktion sowie den Pointer auf den Transaktionspuffer in der Mapped Memory Region des Zielprozesses.

Als nächstes wird der Transaktionspuffer in der Shared Memory Region alloziert und eine Datenstruktur vom Typ `binder_buffer` in der `binder_transaction`-Struktur gespeichert. Die Nutzdaten der Transaktion werden, wie in Listing 5.8 gezeigt, aus dem Adressraum des Absenderprozesses direkt in den Transaktionspuffer kopiert. Nach dem Kopieren der Daten erfolgt die in Abschnitt 4.5 beschriebene Behandlung der aktiven Binder-Objekte. Die Anpassung der Objekte erfolgt auf den kopierten Daten im Transaktionspuffer.

```
1  t->buffer = binder_alloc_buf(target_proc, tr->data_size,
2    tr->offsets_size, !reply && (t->flags & TF_ONE_WAY));
3  [ ... ]
4  t->buffer->allow_user_free = 0;
5  t->buffer->debug_id = t->debug_id;
6  t->buffer->transaction = t;
7  t->buffer->target_node = target_node;
8  [ ... ]
9  if (copy_from_user(t->buffer->data, tr->data.ptr.buffer, tr->data_size)) {
10     [ ... ]
11 }
12 if (copy_from_user(offp, tr->data.ptr.offsets, tr->offsets_size)) {
13     [ ... ]
14 }
```

Listing 5.8: Kopieren der Daten in den Transaktionspuffer

Im letzten Schritt der Transaktionsverarbeitung auf Absenderseite wird die `binder_transaction`-Struktur im Transaktionsstack des Absenderthreads referenziert – sofern es sich nicht um eine One-Way-Transaktion handelt. Weiterhin wird ein Work-Item vom Typ `BINDER_TRANSACTION` in die Todo-Liste des Zielprozesses eingefügt. Dieses Work-Item wird zusätzlich in der `binder_transaction`-Struktur referenziert. In die Todo-Liste des Absenderthreads wird ein Work-Item vom Typ `BINDER_TRANSACTION_COMPLETE` eingefügt. Anschließend wird, wie in Listing 5.9 gezeigt, über das Kommando `wake_up_interruptible()` auf der Waitqueue des Prozesses der nächste freie Looper-Thread reaktiviert um die Transaktion anzunehmen.

```
1  [ ... ]
2  if (reply) {
3    [... ]
4  } else if (!(t->flags & TF_ONE_WAY)) {
5      [... ]
6    t->need_reply = 1;
7    t->from_parent = thread->transaction_stack;
8    thread->transaction_stack = t;
9  } else {
10     [... ]
11 }
12 t->work.type = BINDER_WORK_TRANSACTION;
13 list_add_tail(&t->work.entry, target_list);
14 tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
15 list_add_tail(&tcomplete->entry, &thread->todo);
16 if (target_wait)
17   wake_up_interruptible(target_wait);
18 return;
19 [ ... ]
```

Listing 5.9: Schreiben des Work-Items und Aktivieren des Looper-Threads

Der Absenderthread springt nach dem reaktivieren des Looper-Threads zurück in die Methode `binder_thread_write()` und – sofern keine weiteren Kommandos im Write-

Buffer liegen – zurück in die Methode `binder_ioctl()`. Bei einer One-Way-Transaktion erfolgt der Rücksprung in den Userspace-Prozess. Bei Transaktionen, die eine Antwort erwarten, wird die Methode `binder_thread_read()` aufgerufen. Hier wird der Absenderthread über die Waitqueue des Threads blockiert bis die Antwort vom Service vorliegt.

5.3. Lesen durch einen Looper-Thread

Durch den Aufruf von `wake_up_interruptible()` auf der Waitqueue des Zielprozesses werden die Looper-Threads, die auf eingehende Nachrichten warten, reaktiviert. Der reaktivierte Looper-Thread iteriert über die in der Todo-Liste des Prozesses enthaltenen Work-Items. In der Methode `binder_transaction()` wurde ein Work-Item vom Typ `BINDER_WORK_TRANSACTION` in die Todo-Liste eingefügt. Da das Work-Item in der `transaction_data`-Struktur referenziert ist, können diese über das Makro `container_of`[21] in der Verarbeitung des Looper-Threads geladen werden (Listing Listing 5.10).

```
1   while (1) {
2     uint32_t cmd;
3     struct binder_transaction_data tr;
4     struct binder_work *w;
5     struct binder_transaction *t = NULL;
6     [ ... ]
7     switch (w->type) {
8     case BINDER_WORK_TRANSACTION: {
9           t = container_of(w, struct binder_transaction, work);
10    } break;
11    [ ... ]
12    }
```

Listing 5.10: Ermittlung der `binder_transaction`-Struktur

In Listing 5.11 ist zu sehen, dass in der Verarbeitung im Looper-Thread eine neue Datenstruktur vom Typ `binder_transaction_data` aufgebaut wird. Diese wird mit den Informationen aus der `binder_transaction`-Struktur sowie aus der `binder_node`-Struktur (`target_node` im Listing)angereichert.

[21] Siehe [Pazdera, 2012]

```
1    if (t->buffer->target_node) {
2         struct binder_node *target_node = t->buffer->target_node;
3         tr.target.ptr = target_node->ptr;
4         tr.cookie =  target_node->cookie;
5         [ ... ]
6         cmd = BR_TRANSACTION;
7    } else {
8         tr.target.ptr = NULL;
9         tr.cookie = NULL;
10        cmd = BR_REPLY;
11   }
12   tr.code = t->code;
13   tr.flags = t->flags;
14   tr.sender_euid = from_kuid(current_user_ns(), t->sender_euid);
15   [ ... ]
16   tr.data_size = t->buffer->data_size;
17   tr.offsets_size = t->buffer->offsets_size;
18   tr.data.ptr.buffer = (void *)t->buffer->data +
19                   proc->user_buffer_offset;
20   tr.data.ptr.offsets = tr.data.ptr.buffer +
21                   ALIGN(t->buffer->data_size,
22                       sizeof(void *));
```

Listing 5.11: Anreicherung der `binder_transaction_data`-Struktur

Beim Senden der Daten an den Kernel wurde die Adresse der Datenstruktur `binder_transaction_data` in den Write-Buffer geschrieben und im Kernel aus dem Adressraum des Userspace-Prozesses kopiert. Da der Empfängerprozess die Daten nicht aus dem Adressraum des Kernels kopieren kann, wird, wie in Listing 5.12 gezeigt, die Binder-Response `BR_TRANSACTION` gefolgt von der vollständigen Datenstruktur `binder_transaction_data` in den Read-Buffer des Empfängerthreads geschrieben.

```
1    if (put_user(cmd, (uint32_t __user *)ptr))
2         return -EFAULT;
3    ptr += sizeof(uint32_t);
4    if (copy_to_user(ptr, &tr, sizeof(tr)))
5         return -EFAULT;
6    ptr += sizeof(tr);
```

Listing 5.12: Schreiben des Read-Buffers

5.4. Übergabe an den Service

Im Empfängerprozess erfolgt die Verarbeitung durch den Looper-Thread. Anhand der `binder_transaction_data`-Struktur im Read-Buffer kann auf Empfängerseite aus den Daten in der Mapped Memory Region ein neues Parcel aufgebaut werden (Listing 5.13). Der Service, welcher den Empfänger der Transaktion darstellt wird anhand des Cookies in den Transaktionsdaten identifiziert. Das Cookie gibt die Speicheradresse der `BBinder`-Instanz im Speicher des Userspace-Prozesses an.

Auf den Instanz des `BBinder`-Objektes wird dann die `transact()`-Methode aufgerufen. Die `transact()`-Methode ruft die `onTransact()`-Methode auf, welche die Implementierung der Aktionen auf Empfängerseite enthält.

```
1  status_t IPCThreadState::executeCommand(int32_t cmd)
2  {
3      BBinder* obj;
4  [ ... ]
5      switch (cmd) {
6      [ ... ]
7      case BR_TRANSACTION:
8          {
9              binder_transaction_data tr;
10             result = mIn.read(&tr, sizeof(tr));
11             [ ... ]
12             Parcel buffer;
13             buffer.ipcSetDataReference(
14                 reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
15                 tr.data_size,
16                 reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
17                 tr.offsets_size/sizeof(size_t), freeBuffer, this);
18             mCallingPid = tr.sender_pid;
19             mCallingUid = tr.sender_euid;
20             [ ... ]
21             Parcel reply;
22             [ ... ]
23             if (tr.target.ptr) {
24                 sp<BBinder> b((BBinder*)tr.cookie);
25                 const status_t error = b->transact(tr.code, buffer, &reply,
                       tr.flags);
26                 [ ... ]
27             }
28             [ ... ]
29             if ((tr.flags & TF_ONE_WAY) == 0) {
30                 [ ... ]
31                 sendReply(reply, 0);
32             }
33             [ ... ]
34         }
35         break;
36     [ ... ]
37     }
38     return result;
39 }
```

Listing 5.13: Behandlung einer eingehenden Transaktion

5.5. Antworten auf Transaktionen

Antworten auf Transaktionen werden vom Service mit dem Binder-Kommando `BC_REPLY` verschickt. Dies erfolgt aus der Methode `sendReply()` in der Klasse `IPCThreadState`. Das Verarbeitungsschema einer Antwort ist vergleichbar dem einer Transaktion, die Verarbeitung im Kernel erfolgt ebenfalls über die Methode `binder_transaction()`.

Die Antwort wird durch den Looper-Thread verschickt, der die Transaktion empfangen hat. Im Kernel wurden die Transaktionsdaten beim Schreiben des Read-Buffers im Transaktionsstack des Looper-Threads referenziert. Anhand der Metadaten auf dem Transaktionsstack kann der Absenderthread bestimmt werden. Wie in Listing 5.7 gezeigt, werden in diesem Fall die Todo-Liste und Waitqueue des Threads der die Transaktion ausgelöst hat als Ziele in der Methode `binder_transaction()` verwendet.

```
1  if (reply) {
2    in_reply_to = thread->transaction_stack;
3           [ ... ]
4    thread->transaction_stack = in_reply_to->to_parent;
5    target_thread = in_reply_to->from;
6           [ ... ]
7    }
8    target_proc = target_thread->proc;
9    [ ... ]
```

Listing 57: Ermittlung des Zielthreads bei der Beantwortung einer Transaktion

Die übrige Verarbeitungskette ist annährend identisch zur Transaktion. Da das `wake_up_interruptible()`-Kommando am Ende der Methode `binder_transaction()` auf die Waitqueue des Absenderthreads ausgeführt wird, wird im Absenderprozess kein Looper-Thread fortgesetzt, sondern der Absenderthread der Transaktion.

5.6. Kapitelzusammenfassung

In diesem Kapitel wurde der Ablauf der Transaktionsverarbeitung vom Client bis zum Service beschrieben.

Eine Transaktion wird durch den Aufruf der `transact()`-Methode an einem Binder-Proxy ausgelöst. Hierbei werden die Parcels `data` und `reply` übergeben. Die Daten in den Parcels wurden beim Hinzufügen zum Parcel für die Verarbeitung im Kernel vorbereitet. Die Verarbeitung der Methode `transact()` wird an die gleichnamige Methode der Klasse `IPCThreadState` delegiert, wobei das im `BpBinder`-Objekt gespeicherte Handle übergeben wird. In der Klasse `IPCThreadState` wird der Write-Buffer aufgebaut und im ioctl-Kommando `BINDER_WRITE_READ` an den Kerneltreiber übergeben.

Im Kerneltreiber wird anhand des Handles in den Transaktionsdaten die `binder_ref`-Struktur identifiziert. Diese wird zur Bestimmung des Empfängerprozesses und -services verwendet. Die Transaktionsinhalte werden aus dem Speicherbereich des Absenderprozesses in einen Transaktionspuffer in der Mapped Memory Region des Empfängerprozesses kopiert. Ggf. werden die aktiven Binder-Objekte im Transaktionspuffer des Empfängerprozesses manipuliert. Über die Waitqueue des Empfängerprozesses wird einer der verfügbaren Looper-Threads aktiviert. Bei bidirektionalen Transaktionen wird der Absenderthread in der Methode binder_thread_read() blockiert. In der Verarbeitung der Methode `binder_read_thread()` des Looper-Threads wird der Read-Buffer für den Empfängerprozess aufgebaut und in den Adressraum des Userspace-Prozesses kopiert. Der Read-Buffer enthält die Transaktionsdaten aber nicht die Transaktionsinhalte. In den Transaktionsdaten ist das Cookie aus der `binder_node`-Struktur des angesprochenen Services enthalten.

Im Empfängerprozess wird das Cookie in der Klasse `IPCThreadState` zur Identifizierung des Binder-Objektes, welches den Service implementiert, verwendet. Die Transaktionsinhalte werden aus dem Transaktionspuffer in der Mapped Memory Region in den Adressraum des Empfängerprozesses kopiert und zur Instanziierung des `data`-Parcels verwendet. Am nativen Service wird die `transact()`-Methode aufgerufen, in deren Verarbeitungskette die `onTransact()`-Methode des Java-Implementierung aufgerufen wird.

Antworten auf Transaktionen werden nach dem gleichen Schema verarbeitet, wie die Transaktion selbst. Im Unterschied zur Transaktion wird die Antwort direkt an den Absenderthread geschickt und nicht durch einen Looper-Thread empfangen.

6. Verbindungsaufbau zu Services

Um einen Service nutzen zu können, muss ein Client zuvor einen Binder-Proxy für den Service bekommen. Die zugrundeliegenden Mechanismen für die Übertragung von aktiven Binder-Objekten wurden in den vorangegangenen Kapiteln beschrieben.

In diesem Kapitel wird der Verbindungsaufbau zu einem Service in der Android-Runtime-Umgebung beschrieben. Dies erfolgt nicht mittels der Namensauflösung am Servicemanager sondern über den ActivityManagerService im Android-Systemserver. Hierzu wird vorab beschrieben, wie im Startprozess der App die hierfür erforderliche Verbindung zum ActivityManager Service hergestellt wird.

6.1. Start der App

Alle Apps brauchen eine Verbindung zum ActivityManager-Service, um Services aus anderen Prozessen nutzen zu können. Ebenso hat der ActivityManager-Service Verbindungen zu allen Apps. Diese Verbindungen werden beim Start der App aufgebaut.

Alle Prozesse, die in der Dalvik-VM bzw. In der Android-Runtime ausgeführt werden, werden aus der Zygote „geforked“. Die Zygote selbst ist Server, der in der Dalvik-VM ausgeführt wird und einen Socket öffnet auf dem Nachrichten empfangen werden können. Anders als alle Prozesse, die aus der Zygote heraus gestartet werden, stellt die Zygote keine Services über das Binder-Framework bereit.

Die folgenden Beschreibungen sind stark vereinfacht und beschreiben nur die Aspekte, die für die Funktion des Binder-Frameworks relevant sind.

6.1.1. Fork und Initialisierung der Runtime

Der Start einer App wird vom ActivityManager-Service im Systemserver ausgelöst, indem dieser eine Nachricht mit der Startparametern für die zu startende App an die Zygote schickt. Diese führt daraufhin den `fork()`-Systemcall aus.

Nach dem Fork aus der Zygote entspricht die Laufzeitumgebung des neuen Kindprozesses Umgebung der Zygote. Unmittelbar nach dem Fork wird daher im Kindprozess die Methode `zygoteInit()` in der Klasse `Zygote` aufgerufen, welche die Laufzeitumgebung für die Ausführung einer App vorbereitet. Ein wesentlicher Bestandteil dieser Vorbereitung ist die Instanziierung des prozessweiten Singletons `ProcessState`[22] und dem damit verbundenen Öffnen des Device-Files `/dev/binder` (siehe Abschnitt 3.3.2).

```
1  virtual void onZygoteInit()
2      {
3          // Re-enable tracing now that we're no longer in Zygote.
4          atrace_set_tracing_enabled(true);
5
6          sp<ProcessState> proc = ProcessState::self();
7          ALOGV("App process: starting thread pool.\n");
8          proc->startThreadPool();
9      }
```

Listing 6.1: Instanziierung von `ProcessState` und Start des Threadpools

Listing 6.1 zeigt die Instanziierung des `ProcessState`-Singletons und den anschließenden Start des Threadpools. Der Threadpool umfasst alle Looper-Threads über die Nachrichten vom Kernel empfangen werden können. Für jeden Looper-Thread wird ein Objekt vom Typ `PoolThread` instanziiert. Jeder Pool-Thread instanziiert ein Objekt vom Typ `IPCThreadState` auf dem die Methode `joinThreadPool()` aufgerufen wird um den Thread beim Kerneltreiber als Looper-Thread zu registrieren.

Um den Thread beim Kernel zu registrieren, wird in der Methode `joinThreadPool()` das Binder-Kommando `BC_ENTER_LOOPER` in den Write-Buffer geschrieben und über die Methode `getAndExecuteCommand()` an den Kerneltreiber übergeben.

[22] Beim Start der Zygote wird dieses Objekt nicht instanzziert, daher ist das Binder-Framework in der Zygote nicht nutzbar.

```
1  void IPCThreadState::joinThreadPool(bool isMain)
2  {
3      LOG_THREADPOOL("**** THREAD %p (PID %d) IS JOINING THE THREAD POOL\n",
   (void*)pthread_self(), getpid());
4
5      mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
6      [ ... ]
7      status_t result;
8      do {
9          [ ... ]
10         result = getAndExecuteCommand();
11         [ ... ]
12     } while (result != -ECONNREFUSED && result != -EBADF);
13     [ ... ]
14     mOut.writeInt32(BC_EXIT_LOOPER);
15     talkWithDriver(false);
16 }
```

Listing 6.2: Ausschnitt aus der Methode `joinThreadPool()`

Da das Kommando `BC_ENTER_LOOPER` der ersten Nutzung des Kerneltreiber durch den Thread geschickt wird, wird in der `binder_thread`-Struktur im Kernel bei diesem Aufruf das Flag `BINDER_LOOPER_MUST_RETURN` gesetzt. Dies führt dazu, dass der Thread nin der Methode `binder_thread_read()` nicht blockiert wird. Da außer dem bisher gestarteten Thread keine weiteren Looper-Threads registriert sind und der Thread nicht blockiert wurde, wird der Zähler `threads_available` an der `binder_proc`-Struktur auf 0 gesetzt. Hierdurch wird vor dem Rücksprung aus der Methode `binder_thread_read()` in die Methoden `binder_ioctl()` die Binder-Response `BR_SPAWN_LOOPER` in den Read-Buffer des Threads geschrieben. Vor dem Rücksprung in den Userspace-Prozess wird das Flag `BINDER_LOOPER_STATE_NEED_RETURN` gelöscht. Damit kann der Thread beim nächsten ioctl-Aufruf blockiert werden und Nachrichten empfangen. Entsprechend wird der Zähler `threads_available` auf 1 gesetzt, so dass keine weiteren Threads angefordert werden.

```
1  static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long
   arg)
2  {
3      int ret;
4      struct binder_proc *proc = filp->private_data;
5      struct binder_thread *thread;
6      [ ... ]
7      if (thread)
8         thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
9      binder_unlock(__func__);
10     wait_event_interruptible(binder_user_error_wait,
           binder_stop_on_user_error < 2);
11     [ ... ]
12     return ret;
13 }
```

Listing 6.3: Löschen des `BINDER_LOOPER_NEED_RETURN`-Flags

Da die Response `BR_SPAWN_LOOPER` in den Read-Buffer geschrieben wurde, wird im Userspace-Prozess ein neuer Looper-Thread gestartet. Dieser wird beim Kernel mit dem Binder-Kommando `BC_REGISTER_LOOPER` registriert (siehe Listing 6.2). Hierdurch wird der Zähler `requested_threads` bzw. `requested_threads_started` an der `binder_proc`-Struktur erhöht.

```
1  case BR_SPAWN_LOOPER:
2          mProcess->spawnPooledThread(false);
3          break;
```

Listing 6.4: Start des eines vom Kernel angeforderten Looper-Threads

Nach der Initialisierung des Threadpools sind in der App drei Threads gestartet:

- Mainthread, in dem die App gestartet wird (`ActivityThread`)
- Looper-Thread, der vom Userspace-Prozess gestartet wurde (`Binder_1`)
- Looper-Thread, der vom Kernel angefordert wurde (`Binder_2`)

6.1.2. Verbindung zum ActivityManager Service

Nach der Initialisierung der Looper-Threads wird die App in der Android-Laufzeitumgebung über die `main()`-Methode der Klasse `ActivityThread` gestartet. In der `main()`-Methode wird ein Objekt vom Typ `ActivityThread` instanziiert, auf dem die `attach()`-Methode ausgeführt wird.

```
1  private void attach(boolean system) {
2      sCurrentActivityThread = this;
3      mSystemThread = system;
4      if (!system) {
5          [ ... ]
6          RuntimeInit.setApplicationObject(mAppThread.asBinder());
7          IActivityManager mgr = ActivityManagerNative.getDefault();
8          try {
9              mgr.attachApplication(mAppThread);
10         } catch (RemoteException ex) {
11             // Ignore
12         }
13         [ ... ]
14     }
15     [ ... ]
16 }
```

Listing 6.5: Methode `attach()` der Klasse `ActivityThread`

In Listing 6.5 und Listing 6.6 ist zu sehen, dass in der `attach()`-Methode zunächst beim Servicemanager einen Binder-Proxy für den ActivityManager-Service angefordert wird. Durch den Aufruf der Methode `attachApplication()` wird eine Transaktion mit dem Transaktionscode `ATTACH_APPLICATION_TRANSACTION` an den ActivityManager-Service geschickt. In dieser Transaktion wird das `ApplicationThread`-Object des lokalen Prozessesn an den ActivityManager-Service übergeben. Das `ApplicationThread`-Objekt ist ein Service[23], damit wird dieses Objekt in einer Transaktion wie in Abschnitt 4.5 als aktives Binder-Objekt behandelt. So hat der ActivityManager-Service nach der Ausführung der `ATTACH_APPLICATION_TRANSACTION` einen Binder-Proxy für den Application-Thread der App.

```
1  private static final Singleton<IActivityManager> gDefault = new
   Singleton<IActivityManager>() {
2          protected IActivityManager create() {
3              IBinder b = ServiceManager.getService("activity");
4              if (false) {
5                  Log.v("ActivityManager", "default service binder = " + b);
6              }
7              IActivityManager am = asInterface(b);
8              if (false) {
9                  Log.v("ActivityManager", "default service = " + am);
10             }
11             return am;
12         }
13     };
```

Listing 6.6: Anforderung des Binder-Proxies für den ActivityManager-Service

[23] Das heisst, die Klasse ApplicationThread erbt von der Klasse Binder.

6.2. Binding eines Services

In den folgenden Beispielen ist beschrieben, welche Schritte ausgeführt werden, wenn aus einer App heraus ein Service über die `bindService()`-Methode des gebunden wird. Neben Services können auch Activities, BroadcastReceiver oder MessageReceiver über das Binder-Framework genutzt werden. Da die Grundprinzipien bei allen Komponenten gleich sind, steht der Service hier exemplarisch für alle anderen Komponenten.

6.2.1. Binden in der App (Client)

Die Adressierung des Services erfolgt über einen Intent[24]. Innerhalb der App wird der Service an ein Objekt vom Typ `ServiceConnection` gebunden. Das Binden des Services erfolgt durch Aufruf der Methode `bindService()`, die vom Applikationskontext bereitgestellt wird. Hierbei werden der Intent und das `ServiceConnection`-Objekt als Argumente an die Methode `bindService()` übergeben.

```
1  public class MyActivity extends ActionBarActivity {
2
3
4    ServiceConnection mServiceConnection = new MyServiceConnection();
5
6    public void getRemotePid(View view){
7          Intent intent = new Intent();
8          intent.setClassName("de.coldspot.android.aidlserviceexample",
               "de.coldspot.android.aidlserviceexample.getPidService");
9          boolean bound = bindService(intent, mServiceConnection,
               Context.BIND_AUTO_CREATE);
10     }
11 }
```

Listing 6.7: Beispielhafter Aufruf zum Binden eines Services

Die für das Binden des Service relevante Implementierung ist in der Methode `bindServiceCommon()` in der Klasse `ContextImpl` enthalten. Hier wird die Methode `bindService()` auf dem Binder-Proxy für den ActivityManager-Service[25] aufgerufen.

---

[24] Zur Erklärung eines Intents, siehe Anhang A.1.

[25] Die Methode `ActivityManagerNative.getDefault()` liefert in einer App immer den Proxy für den ActivityManagerService zurück.

```
1  private boolean bindServiceCommon(Intent service, ServiceConnection conn, int
   flags, UserHandle user) {
2          IServiceConnection sd;
3
4              [ ... ]
5
6              service.prepareToLeaveProcess();
7              int res = ActivityManagerNative.getDefault().bindService(
8                  mMainThread.getApplicationThread(), getActivityToken(),
9                  service, service.resolveTypeIfNeeded(getContentResolver()),
10                 sd, flags, user.getIdentifier());
11             if (res < 0) {
12                 throw new SecurityException(
13                         "Not allowed to bind to service " + service);
14             }
15             return res != 0;
16         } catch (RemoteException e) {
17             return false;
18         }
19     }
```

Listing 6.8: Aufruf von `bindService()`

Die Methode `bindService()` ist ein Wrapper um die `transact()`-Methode des Binder-Proxies für den ActivityManager-Service. Listing 6.9 zeigt, wie die Parcels `data` und `reply` aufgebaut werden und die `transact()`-Methode mit dem Transaktionscode `BIND_SERVICE_TRANSACTION` ausgeführt werden.

```
1  public int bindService(IApplicationThread caller, IBinder token,
2              Intent service, String resolvedType, IServiceConnection
   connection,
3              int flags, int userId) throws RemoteException {
4          Parcel data = Parcel.obtain();
5          Parcel reply = Parcel.obtain();
6          data.writeInterfaceToken(IActivityManager.descriptor);
7          data.writeStrongBinder(caller != null ? caller.asBinder() : null);
8          data.writeStrongBinder(token);
9          service.writeToParcel(data, 0);
10         data.writeString(resolvedType);
11         data.writeStrongBinder(connection.asBinder());
12         data.writeInt(flags);
13         data.writeInt(userId);
14         mRemote.transact(BIND_SERVICE_TRANSACTION, data, reply, 0);
15         reply.readException();
16         int res = reply.readInt();
17         data.recycle();
18         reply.recycle();
19         return res;
20     }
```

Listing 6.9: Ausführen der `BIND_SERVICE_TRANSACTION`

Das data-Parcel enthält drei aktive Binder-Objekte, die vom ActivityManager-Service verwendet werden:

- `caller`: Binder auf das `ApplicationThread`-Objekt der App

- `token`: Token für die aktuelle Activity, wird als Security-Token verwendet[26]
- `connection`: Binder-Objekt für die `ServiceConnection`, welche den Service in der App bereitstellt

6.2.2. Verarbeitung im ActivityManager-Service

Im ActivityManager-Service wird die Transaktion in der `onTransact()`-Methode durch den in Listing 6.10 angegebenen Codeblock behandelt. Hier ist zu sehen, wie die Daten und aktiven Binder-Objekte aus dem `data`-Parcel werden. Hierbei ist zu beachten, dass in der App Objekte vom Typ `Binder` in das Parcel geschrieben wurden. Durch die in Abschnitt 4.5.3.1 beschriebene Behandlung von aktiven Binder-Objekten werden hier Binder-Proxies erzeugt. Die Verarbeitung wird an die Methode `bindService()` in der Klasse `ActivityManagerService` delegiert.

```
1  case BIND_SERVICE_TRANSACTION: {
2      data.enforceInterface(IActivityManager.descriptor);
3      IBinder b = data.readStrongBinder();
4      IApplicationThread app = ApplicationThreadNative.asInterface(b);
5      IBinder token = data.readStrongBinder();
6      Intent service = Intent.CREATOR.createFromParcel(data);
7      String resolvedType = data.readString();
8      b = data.readStrongBinder();
9      int fl = data.readInt();
10     int userId = data.readInt();
11     IServiceConnection conn = IServiceConnection.Stub.asInterface(b);
12     int res = bindService(app, token, service, resolvedType, conn, fl,
           userId);
13     reply.writeNoException();
14     reply.writeInt(res);
15     return true;
16 }
```

Listing 6.10: Behandlung der `BIND_SERVICE_TRANSACTION`

In der Verarbeitungskette der `bindService()`-Methode werden viele Einzelschritte ausgeführt, die hier nicht näher beschrieben werden, hierzu zählen u.a.:

- Auflösen des Services anhand des von der App gelieferten Intents

[26] Die Funktion zur Nutzung von Binder-Objekten als SecurityToken ist in dieser Arbeit nicht berücksichtigt

- Authentisierung der Transaktion anhand den gelieferten Tokens, der UID in der Transaktion und des Absenderthreads
- Starten des Services sofern erforderlich
- Registrierung des Services und Erzeugung von Datenstrukturen zur Verwaltung des Services und der `ServiceConnection`

Um das Binding des Services durchzuführen wird über den Binder-Proxy für den Application-Thread des Serviceproviders eine Transaktion vom Typ `SCHEDULE_BIND_SERVICE` an den Service geschickt. Im `data`-Parcel dieser Transaktion wird das `ServiceRecord`-Objekt in dem der Service im ActivityManager-Service verwaltet wird, übergeben. Das `ServiceRecord`-Objekt ist ein Service, damit wird beim ServiceProvider ein Binder-Proxy für das Objekt instanziiert. Da nur das `ServiceRecord`-Objekt an den Serviceprovider geschickt wird, bekommt dieser keine direkte Referenz auf den anfragenden Prozess. Wie in Listing 6.11 zu sehen, wird die Transaktion als One-Way Transaktion an den Serviceprovider geschickt, der ActivityManager-Service erwartet also keine Antwort.

```
1   public final void scheduleBindService(IBinder token, Intent intent, boolean
    rebind,
2               int processState) throws RemoteException {
3           Parcel data = Parcel.obtain();
4           data.writeInterfaceToken(IApplicationThread.descriptor);
5           data.writeStrongBinder(token);
6           intent.writeToParcel(data, 0);
7           data.writeInt(rebind ? 1 : 0);
8           data.writeInt(processState);
9           mRemote.transact(SCHEDULE_BIND_SERVICE_TRANSACTION, data, null,
10                  IBinder.FLAG_ONEWAY);
11          data.recycle();
12      }
```

Listing 6.11: Auslösen der `SCHEDULE_BIND_SERVICE_TRANSACTION`

6.2.3. Binden des Services im Server-Prozess

Im Serviceprovider wird die `SCHEDULE_BIND_SERVICE_TRANSACTION` -Transaktion in der `onTransact()`-Methode des `ApplicationThread`-Objektes verarbeitet. Die Ausführung wird an die Methode `scheduleBindService()` delegiert.

```
1  case SCHEDULE_BIND_SERVICE_TRANSACTION: {
2      data.enforceInterface(IApplicationThread.descriptor);
3      IBinder token = data.readStrongBinder();
4      Intent intent = Intent.CREATOR.createFromParcel(data);
5      boolean rebind = data.readInt() != 0;
6      int processState = data.readInt();
7      scheduleBindService(token, intent, rebind, processState);
8      return true;
9  }
```

Listing 6.12: Behandlung der `SCHEDULE_BIND_TRANSACTION`

In der Ausführungshierarchie der `scheduleBindService()`-Methode wird eine interne Nachricht an den Activity-Thread geschickt, in dem die Nachricht durch die `handleBindService()`-Methode verarbeitet wird. In der Nachricht wird der vom ActivityManager-Service gelieferte Binder-Proxy verwendet, um den zu bindenden Service zu bestimmen. Durch den Aufruf der `onBind()`-Methode[27] für den Service wird das Service-Objekt im Serviceprovider instanziiert.

```
1  private void handleBindService(BindServiceData data) {
2     Service s = mServices.get(data.token);
3     [ ... ]
4     if (s != null) {
5          try {
6              data.intent.setExtrasClassLoader(s.getClassLoader());
7              try {
8                  if (!data.rebind) {
9                      IBinder binder = s.onBind(data.intent);
10                     ActivityManagerNative.getDefault().publishService(
11                             data.token, data.intent, binder);
12                 } else {
13                     s.onRebind(data.intent);
14                     ActivityManagerNative.getDefault().serviceDoneExecuting(
                          data.token, 0, 0, 0);
15                 }
16                 ensureJitEnabled();
17             } catch (RemoteException ex) {
18                 [ ... ]
19             }
20         } catch (Exception e) {
21             [ ... ]
22         }
23     }
24 }
```

Listing 6.13: Methode `handleBindService()` in `ActivityThread`

Nach der Instanziierung des Services wird die `publishService()`-Methode auf dem Binder-Proxy für den ActivityManager-Service im ServiceProvider aufgerufen. Hierdurch

[27] Der Fall des rebindings wird hier nicht betrachtet.

wird eine Transaktion mit dem Transaktionscode `PUBLISH_SERVICE_TRANSACTION` an den ActivityManager-Service geschickt. Hierbei werden u.a. das Token (Binder-Proxy für den `ServiceRecord`) und der Service übergeben.

```
1  public void publishService(IBinder token,
2          Intent intent, IBinder service) throws RemoteException {
3      Parcel data = Parcel.obtain();
4      Parcel reply = Parcel.obtain();
5      data.writeInterfaceToken(IActivityManager.descriptor);
6      data.writeStrongBinder(token);
7      intent.writeToParcel(data, 0);
8      data.writeStrongBinder(service);
9      mRemote.transact(PUBLISH_SERVICE_TRANSACTION, data, reply, 0);
10     reply.readException();
11     data.recycle();
12     reply.recycle();
13 }
```

Listing 6.14: Ausführen der `PUBLISH_SERVICE_TRANSACTION`

6.2.4. Übergabe des Services an die App im ActivityManagerService

Im ActivityManager-Service werden die vom Serviceprovider gelieferten aktiven Binder-Objekte aus dem Parcel ausgelesen. Hierbei ist zu beachten, dass das `token`, welches vom Serviceprovider übergeben wurde, ein Binder-Proxy für den `ServiceRecord` im ActivityManager-Service ist. Binder-Proxies, die an ihren Serviceprovider[28] geschickt werden, werden aufgrund der in Abschnitt 4.5.3.2 beschriebenen Behandlung im Kernel, im Empfängerprozess in den vom Proxy refrenzierten Service aufgelöst.

Die Behandlung der Transaktion erfolgt im ActivityManager-Service in der Methode `publishServiceLocked()`. Hier werden anhand des `ServiceRecord`-Objektes alle `ServiceConnections` bestimmt, welche den Service gebunden haben. Im `ServiceRecord`-Objekt sind alle Binder-Proxies für die `ServiceConnection`-Objekte in den den Apps, die den Service gebunden haben, referenziert. Auf jedem dieser Binder-Proxies wird die `connect()`-Methode aufgerufen, wobei der vom Serviceprovider gelieferte Binder-Proxy für den Service übergeben wird.

[28] In diesem Falle ist der ActivityManager-Service der Serviceprovider des `ServiceRecord`-Objektes

```
1  void publishServiceLocked(ServiceRecord r, Intent intent, IBinder service) {
2          final long origId = Binder.clearCallingIdentity();
3          try {
4              [ ... ]
5              if (r != null) {
6                  [ ... ]
7                  if (b != null && !b.received) {
8                      [ ... ]
9                      for (int conni=r.connections.size()-1; conni>=0; conni--)
   {
10                         ArrayList<ConnectionRecord> clist =
11                             r.connections.valueAt(conni);
12                         for (int i=0; i<clist.size(); i++) {
13                             ConnectionRecord c = clist.get(i);
14                             [ ... ]
15                             try {
16                                 c.conn.connected(r.name, service);
17                             } catch (Exception e) {
18                             [ ... ]
19                             }
20                         }
21                     }
22                 }
23             [ ... ]
24             }
25         } finally {
26             Binder.restoreCallingIdentity(origId);
27         }
28     }
```

Listing 6.15: Methode `publishServiceLocked()`

6.2.5. Binden des Services in der ServiceConnection in der App

Die `connect()`-Methode schickt eine Transaktion an das `ServiceConnection`-Objekt in der App. Hierbei wird der Binder-Proxy für den Service im `data`-Parcel übergeben. Da Binder-Proxies in einer Transaktion an einen dritten Prozess nicht verändert werden, kommt in der App ein Binder-Proxy an.

```
1  @Override public void connected(android.content.ComponentName name,
       android.os.IBinder service) throws android.os.RemoteException
2  {
3    android.os.Parcel _data = android.os.Parcel.obtain();
4    try {
5          _data.writeInterfaceToken(DESCRIPTOR);
6          if ((name!=null)) {
7              _data.writeInt(1);
8              name.writeToParcel(_data, 0);
9          }
10         else {
11             _data.writeInt(0);
12         }
13         _data.writeStrongBinder(service);
14         mRemote.transact(Stub.TRANSACTION_connected, _data, null,
               android.os.IBinder.FLAG_ONEWAY);
15   }
16   [ ... ]
17 }
```

Listing 6.16: Methode `connect()`

Durch die Transaktion wird auf dem ServiceConnection-Objekt in der App die Methode `onServiceConnected()` aufgerufen. Dabei wird der Binder-Proxy für den angefragten Service als Argument übergeben. Diese Methode muss vom Entwickler selbst implementiert werden.

6.3. Kapitelzusammenfassung

In diesem Kapitel wurde der Verbindungsaufbau zu einem Service über den ActivityManagerService und die hierfür erforderlichen Initialisierungen beim App-Start beschrieben.

Beim Start einer App wird nach dem Fork aus der Zygote der Threadpool initialisiert. Hierbei werden für jede App zwei Looper-Threads gestartet. Anschließend wird über den Servicemanager ein Binder-Proxy für den ActivityManager-Service angefordert. Mittels des Binder-Proxies wird dem ActivityManager-Service ein der Service für den Acitivity-Thread der neu gestarteten App übergeben. Damit hat Jede App nach dem Start eine Verbindung zum ActivityManager-Service und der ActivityManager-Service eine Verbindung zur App.

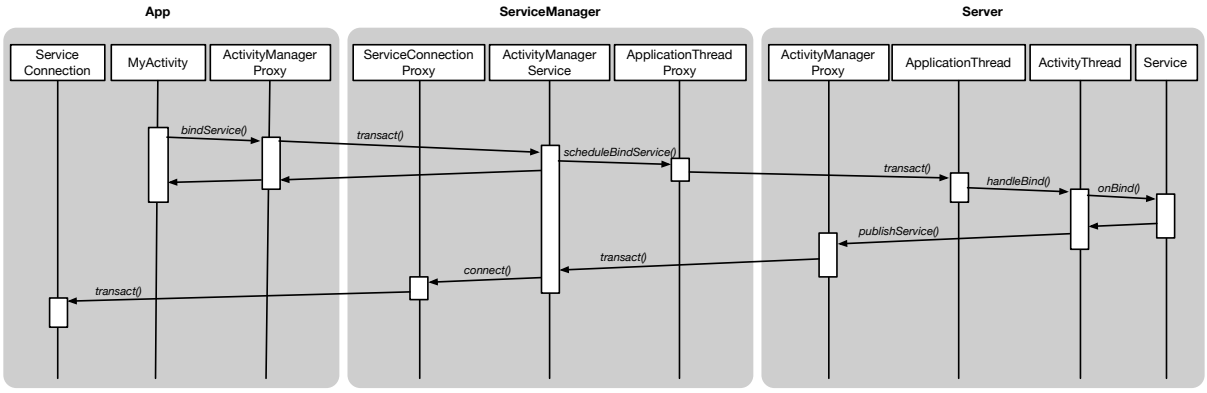

Abbildung 6.1: Vereinfachte Darstellung des Bindens eine Services

Die beim Start aufgebauten Verbindungen werden beim Binden eines Services genutzt um die `BIND_SERVICE`-Transaktion vom Client an den ActivityManager-Service zu schicken. Im Zuge des Verbindungsaufbaus zum Service wird die Verbindung vom ActivityManager-Service zum Aplication-Thread des Serviceproviders verwendet. Der Serviceprovider liefert den Service an den ActivityManager-Service zurück, der dies an die vom Client bereitgestellte `ServiceConnection` übergibt.

Der Verbindungsaufbau erfolgt über eine Serie von Transaktionen zwischen Client, ActivityManager-Service und Serviceprovider. Eine vereinfachte Darstellung des Ablaufs ist in Abbildung 6.1 angegeben. In jeder Transaktion werden aktive Binder-Objekte zwischen den einzelnen Prozessen ausgetauscht. Nach Abschluss des Verbindungsaufbaus hält die `ServiceConnection` im Client-Prozess einen Binder-Proxy, über den die direkte Kommunikation mit dem Service möglich ist.

7. Sicherheitsbetrachtungen

Das Binder-Framework ermöglicht den Zugriff auf Apps, die Zugriff auf sensible Informationen, wie Passworte oder personenbezogene Daten haben. Weiterhin ermöglicht es die Nutzung von Systemkomponenten, die mit erhöhten Berechtigungen im Linux-System ausgeführt werden. Darüber hinaus basiert ein Teil der Sicherheitsarchitektur der zentralen Android-Services auf den Mechanismen des Binder-Frameworks.

In den folgenden Abschnitten sind verschiedene sicherheitsrelevante Aspekte beschrieben, die Gegenstand weiterführender Untersuchungen sein können.

7.1. Adressierung

Die Adressierung von Services basiert auf den Speicheradressen der `binder_node`-Strukturen, die in den `binder_ref`-Strukturen am Client-Prozess gespeichert sind. In einem möglichen Angriffsszenario könnte ein Angreifer versuchen, die Speicheradresse in einer `binder_ref`-Struktur so verändern, dass diese auf einen gefälschten Service zeigt.

Die `binder_ref`-Strukturen werden im Adressraum des Kernels gespeichert. Im Userspace-Prozess werden die Strukturen über prozesslokale Handles referenziert, so dass die Speicheradresse der `binder_ref`-Struktur im Userspace-Prozess nicht bekannt ist. Ein direkter Zugriff auf die `binder_ref`-Strukturen oder die Manipulation eben dieser ist über die vorhandenen ioctl- und Binder-Kommandos nicht möglich.

Für die Manipulation der `binder_ref`-Strukturen ist daher ein direkter schreibender Zugriff auf den Adressraum des Kernels erforderlich.

7.2. Sicherheitsziele bei Transaktionsinhalten

In Transaktionen werden sensible Informationen, wie personenbezogene Daten und Zugangsdaten, an andere Prozesse übertragen. Daher ist die Vertraulichkeit, Integrität und Authentizität in der Transaktionsverarbeitung besonders wichtig. Durch einen erfolgreichen Angriff auf die Transaktionsverarbeitung können sensible Daten ausgespäht, verän-

dert oder gefälscht werden. In der Transaktionsverarbeitung müssen daher Authentizität, Integrität und Vertraulichkeit der Transaktionsinhalte gewährleistet werden.

7.2.1. Authentizität

Die Authentizität einer Transaktion kann anhand des Absenders bestimmt werden. Hierzu wird in der Transaktion die UID des Absenders an den Empfänger übergeben. Die UID kann im Absenderprozess gesetzt werden, diese wird aber durch das Kerneltreiber überschrieben. So ist für den Empfänger der Transaktion die Authentizität gewährleistet.

7.2.2. Integrität

Für die in einer Transaktion übertragenen Daten wird keine Checksumme zur Verifikation auf Empfängerseite generiert. Es ist also kein formaler Mechanismus zur Prüfung der Integrität der Transaktionsinhalte implementiert.

Die Transaktionsinhalte werden in einer atomaren Operation aus dem Adressraum des Absenderprozesses direkt in Transaktionspuffer des Empfängerprozesses kopiert, es erfolgt keine Übertragung über eine dritte Komponente oder Zwischenspeicher. Der Schreibzugriff auf den Transaktionspuffer kann nur durch den Kernel erfolgen. Somit ist eine Manipulation der Transaktionsinhalte nur vor der Kopieroperation im Adressraum des Absenderprozesses oder nach dem Kopieren aus dem Transaktionspuffer in den Adressraum des Empfängerprozesses möglich. Beides setzt einen erfolgreichen Angriff auf die jeweilige App voraus.

7.2.3. Vertraulichkeit

Transaktionsinhalte werden, wie im vorigen Abschnitt beschrieben, direkt aus dem Adressraum des Absenderprozesses in den Transaktionspuffer des Empfängerprozesses geschrieben. Der Transaktionspuffer liegt in einem Speicherbereich im Adressraum des Kernels, der in den Adressraum des Empfängerprozesses gemappt wird. Ein Zugriff auf den Transaktionspuffer durch andere Prozesse ist nicht möglich.

7.3. SELinux / SEAndroid

Wie in Abschnitt 2.4.4 beschrieben, ist das SELinux-Framework Bestandteil von Android, wird hier aber als SEAndroid-Framework bezeichnet. Dieses Framework ermöglicht die regelbasierte Prüfung und Durchsetzung von Zugriffsberechtigungen auf Kernel-Ebene. Im Kerneltreiber ist die Nutzung des SEAndroid-Frameworks in den folgenden Anwendungsfällen berücksichtigt.

- Registrierung des Context-Managers (Listing 7.1)
- Transaktionsübertragung (Listing 7.2)
- Übertragung eines aktiven Binder-Objektes (Listing 7.3)
- Übertragung eines File-Deskriptors (Listing 7.4)

In den vorhandenen Implementierungen werden die `task_struct`-Strukturen der jeweils beteiligten Prozesse an die Prüfroutine übergeben. Hiermit sind Prüfungen auf Prozessebene und Filter anhand der im `task_struct` vorhandenen Informationen zum Prozess möglich. Prüfungen auf Basis der Transaktionseigenschaften, wie z.B. des Transaktionscodes, sind nicht vorgesehen. Ein Konzept für eine weiter führende Protokollierung und Einschränkung von Transaktionen ist beispielsweise mit der in [Hausner, 2014] beschriebenen „binderwall" möglich.

```
1   ret = security_binder_set_context_mgr(proc->tsk);
1   if (ret < 0)
2     goto err;
```

Listing 7.1: AndroidSE-Behandlung für SET_CONTEXT_MGR

```
1   if (security_binder_transaction(proc->tsk, target_proc->tsk) < 0) {
2     return_error = BR_FAILED_REPLY;
3     goto err_invalid_target_handle;
4   }
```

Listing 7.2: AndroidSE-Behandlung für Transaktionen

```
5   if (security_binder_transfer_binder(proc->tsk, target_proc->tsk)) {
6     return_error = BR_FAILED_REPLY;
7     goto err_binder_get_ref_failed;
8   }
```

Listing 7.3: AndroidSE-Behandlung für Binder-Objekte

```
1  if (security_binder_transfer_file(proc->tsk, target_proc->tsk, file) < 0) {
2    fput(file);
3    return_error = BR_FAILED_REPLY;
4    goto err_get_unused_fd_failed;
5  }
```

Listing 7.4: AndroidSE-Behandlung für File-Desriptoren

7.4. Servicemanager

Sofern keine Zugriffsbeschränkungen auf Basis der SEAndroid-Infrastruktur implementiert sind, kann jeder Prozess Transaktionen an jeden Service schicken. Hierzu muss der Prozess über einen der Naming-Services einen `BinderProxy` anfordern. Die Naming-Services bieten hierbei keine Möglichkeit zur Beschränkung der Namensauflösung[29].

Bei den Naming-Services stehen in der Sicherheitsbetrachtung die Gewährleistung der Authentizität der Services und die Integrität der lokalen Servicedatenbank im Vordergrund. Durch die Kompromittierung wäre es möglich, Applikations- oder Systemservices durch kompromittierte Dienste auszutauschen.

Der NamingService des PackageManagers ist aufgrund des Umfangs der hierfür erforderlichen Betrachtung nicht Bestandteil dieser Arbeit.

7.4.1. Kompromittierung des Prozesses

Um einen Austausch des Context-Managers zur Laufzeit zu verhindern, ist es nicht möglich, das ioctl-Kommando `BINDER_SET_CONTEXT_MANAGER` mehrfach auszuführen. Ist ein Context-Manager beim Kernel registriert, dann ist wird der Aufruf dieses ioctl-Kommandos vom Kernel mit einem Fehlercode beantwortet. Weiterhin ist es nicht möglich, den Servicemanager zu deregistrieren. Eine Kompromittierung des Servicemanagers wäre daher nur Austausch des Programms oder durch Manipulation der Init-Konfiguration

[29] Da Prozesse untereinander Binder-Proxies austauschen können, wäre eine Einschränkung der Namensauflösung nur eingeschränkt wirksam.

möglich – beides erfordert Root-Zugriff im Linux-System sowei schreibenden Zugriff auf die Systempartition.

```
1  static struct {
2      unsigned uid;
3      const char *name;
4  } allowed[] = {
5      { AID_MEDIA, "media.audio_flinger" },
6      { AID_MEDIA, "media.log" },
7      { AID_MEDIA, "media.player" },
8      { AID_MEDIA, "media.camera" },
9      { AID_MEDIA, "media.audio_policy" },
10     { AID_DRM,   "drm.drmManager" },
11     { AID_NFC,   "nfc" },
12     { AID_BLUETOOTH, "bluetooth" },
13     { AID_RADIO, "radio.phone" },
14     { AID_RADIO, "radio.sms" },
15     { AID_RADIO, "radio.phonesubinfo" },
16     { AID_RADIO, "radio.simphonebook" },
17 /* TODO: remove after phone services are updated: */
18     { AID_RADIO, "phone" },
19     { AID_RADIO, "sip" },
20     { AID_RADIO, "isms" },
21     { AID_RADIO, "iphonesubinfo" },
22     { AID_RADIO, "simphonebook" },
23     { AID_MEDIA, "common_time.clock" },
24     { AID_MEDIA, "common_time.config" },
25     { AID_KEYSTORE, "android.security.keystore" },
26 };
```

Listing 7.5: Liste der erlaubten UIDs im Servicemanager

Der Servicemanager erlaubt die Registrierung von Services nur durch bestimmte UIDs[30], hierdurch wird sichergestellt, dass nur zugelassene Systemservices beim Servicemanager registriert werden. Die erlaubten Services sind im Servicemanager hart kodiert (siehe Listing Listing 7.5). Die Prüfung der UID erfolgt in der Methode `svc_can_register()` (Listing 7.6).

```
1  int svc_can_register(unsigned uid, uint16_t *name)
2  {
3      unsigned n;
4
5      if ((uid == 0) || (uid == AID_SYSTEM))
6          return 1;
7  for (n = 0; n < sizeof(allowed) / sizeof(allowed[0]); n++)
8          if ((uid == allowed[n].uid) && str16eq(name, allowed[n].name))
9              return 1;
10 return 0;
11 }
```

Listing 7.6: Methode `svc_can_register()` im Servicemanager

[30] Dieses Beispiel zeigt, dass die Absender-UID in den Transaktionen ein sicherheitskritisches Merkmal ist.

Wie in Abschnitt 2.6.6 beschrieben, ist die Deregistrierung eines Services nur durch eine Death-Notification möglich. Um also die Registrierung eines Services zu ändern und beispielsweise auf eine kompromittierte Version des Services zeigen zu lassen, muss der jeweilige Service beendet und dann unter der richtigen UID neu gestartet werden.

```
1   int do_add_service(struct binder_state *bs,
2                      uint16_t *s, unsigned len,
3                      void *ptr, unsigned uid, int allow_isolated)
4   {
5       [ ... ]
6       if (!svc_can_register(uid, s)) {
7           ALOGE("add_service('%s',%p) uid=%d - PERMISSION DENIED\n",
8                str8(s), ptr, uid);
9           return -1;
10      }
11
12      si = find_svc(s, len);
13      if (si) {
14          if (si->ptr) {
15              ALOGE("add_service('%s',%p) uid=%d - ALREADY REGISTERED,
    OVERRIDE\n",
16                   str8(s), ptr, uid);
17              svcinfo_death(bs, si);
18          }
19          si->ptr = ptr;
20      [ ... ]
```

Listing 7.7: Methode `do_add_service()` des Servicemanagers

7.5. Memory Mapping und Transaktionspuffer

Transaktionsdaten werden dem Empfängerprozess in Transaktionspuffern zur Verfügung gestellt. Ein erfolgreicher Zugriff auf den Transaktionspuffer durch einen Angreifer hätte folgende Auswirkungen:

- Einschränkung der Vertraulichkeit der Transaktionsdaten
- Gefährdung der Integrität von Transaktionsdaten

Der Speicherbereich für die Transaktionspuffer eines Empfängerprozesses wird im Adressraum des Kernels alloziert und in den Adressraum des jeweiligen Userspace-Prozesses gemappt. Hierbei erhält der Userspace-Prozess nur lesenden Zugriff auf den Speicherbereich.

Die Sicherheit der Implementierung des Memory Mappings und der Verwaltung der Transaktionspuffer ist nicht Bestandteil dieser Arbeit und sollte in weiteführenden Untersuchungen überprüft werden.

7.6. Kapitelzusammenfassung

In diesem Kapitel wurden verschiedene sicherheitsrelevante Aspekte des Binder-Frameworks betrachtet.

Die Sicherheit in der Adressierung von Services sowie Authentizität, Integrität und Vertraulichkeit von Transaktionsinhalten basiert im wesentlichen auf der Sicherheit des Linux-Kernels. Außer dem Absender, dem Empfänger und dem Kerneltreiber sind keine weiteren Komponenten an der Transaktionsverarbeitung beteiligt. In der Verarbeitung der Transaktionen werden die Transaktionsinhalte direkt vom Adressraum des Absenders in einen nur durch den Empfänger lesbaren Speicherbereich kopiert.

Mit Hilfe des SEAndroid-Frameworks, ist es möglich, anhand von ACLs Einschränkungen bezüglich der Nutzung des Binder-Frameworks auf Prozessebene festzulegen.

Der Servicemanager ist eine sicherheitskritische Komponente. Er ist durch verschiedene Mechanismen gegen Manipulation geschützt. Dies umfasst einen frühen Start im Init-Prozess, die Einstufung als kritischer-Prozess im Init-Prozess sowie die Möglichkeit seitens des Kernels nur einen Context-Manager zur Laufzeit zu registrieren. Der Servicemanager selber lässt Registrierungen nur von Systembenutzern zu.

8. Fazit

Android ist mit einem Marktanteil von 83,1% am weltweiten Smartphone-Markt das weitesten verbreitete Betriebssystem für Smartphones und Tablets. Aufgrund der vielfältigen Anwendungsszenarien und der auf einem Smartphone gespeicherten Daten kommt der Sicherheit des Betriebssystems eine besondere Bedeutung zu. Android basiert auf Linux, wurde aber in zentralen Punkten für den Einsatz im mobilen Umfeld angepasst. Einer der wichtigsten Unterschiede zu herkömmlichen Linux-Systemen besteht in der Interprozesskommunikation. Aufgrund des besonderen Lebenszyklus von Android-Apps, würden die unter Linux gängigen Mechanismen zur Interprozesskommunikation die Stabilität des Betriebssystems gefährden. Aus diesem Grund wurde für die Interprozesskommunikation unter Android das Binder-Framework entwickelt.

Obwohl Android unter einer Open Source-Lizenz steht und der Quelltext frei verfügbar ist, ist das Binder-Framework in bisher veröffentlichten Publikationen nur in zusammengefasster Form oder mit Hinblick auf einzelne Teilaspekte beschrieben. Eine umfassende und detaillierte Beschreibung des Frameworks und der Funktionsweise ist bisher nicht öffentlich verfügbar. Das Ziel dieser Arbeit ist daher, die Erstellung einer Dokumentation, die das Framework und die darin implementierten Funktionen detailliert im Gesamtkontext beschreibt. Diese soll als Grundlage und Orientierungshilfe in weiterführende Untersuchungen zur Sicherheit von Android und des Binder-Frameworks genutzt werden können.

Die Kernfunktionalität des Binder-Frameworks ist der transaktionsbasierte Nachrichtenaustausch zwischen Prozessen im Client-Server-Modell. Ein Service wird im Client-Prozess durch ein Binder-Proxy repräsentiert. Eine Transaktion wird durch Aufruf der Methode `transact()` am Binder-Proxy ausgelöst. In Transaktionen können Daten in sog. Parcels an den Service übergeben werden. Binder-Proxies, Parcels und Serviceklassen bilden das Binder-API. Dieses abstrahiert die Funktionen des Frameworks für die Nutzung in Apps. Dias Binder-API ermöglicht, die Nutzung des Frameworks sowohl in Java als auch in C++. Der Großteil der Logik ist hierbei in C++ implementiert. Die Java-Klassen sind

eng an die native Implementierung gekoppelt, so dass bei der Nutzung des Binder-API in Java-Anwendungen auch immer die nativen Implementierungen beteiligt sind.

Die Grundlage des Binder-Frameworks bildet der Kerneltreiber „binder“. Die Kommunikation zwischen Userspace-Prozess und Kerneltreiber erfolgt über das Device-File `/dev/binder` auf Basis eines nachrichtenorientierten Kommunikationsprotokolls. Die Kommunikation mit dem Kernelmodul erfolgt nicht aus den Klassen des Binder-API, sondern aus Klassen der Prozessinfrastruktur aller Apps. Diese abstrahieren das Kommunikationsprotokoll des Binders für die Nutzung im Binder-API.

Im Kerneltreiber werden die Prozesse, Threads, Binder-Proxies und Services in den Userspace-Prozessen durch Datenstrukturen repräsentiert, die prozessbezogen gespeichert werden. Diese bilden die Grundlage der Adressierung von Services im Binder-Framework. Darüber hinaus wird hierdurch die Steuerung der Nachrichtenübertragung zwischen Threads und Prozessen ermöglicht.

Ein Service wird im Client-Prozess anhand eines vom Kerneltreiber erzeugten Handles adressiert. Das Handle identifiziert eine Datenstruktur von Typ `binder_ref` im Kernel. Die `binder_ref`-Struktur verweist auf eine Datenstruktur im Kernel vom Typ `binder_node`. In der `binder_node`-Struktur ist die Speicheradresse des Objektes gespeichert, welches den Service im Prozess des Serviceproviders identifiziert.

Der Nachrichtenaustausch erfolgt zwischen einzelnen Threads, wobei nur Antworten direkt an einen bestimmten Thread adressiert werden. Bei Transaktionen an einen Service ist der Empfängerthread nicht vorab bestimmbar, daher wird die Nachricht an den Prozess adressiert und von einem Looper-Thread angenommen. Um sicher zu stellen, dass eine App in der Lage ist, Nachrichten zu empfangen, kann der Kerneltreiber zusätzliche Looper-Threads beim Userspace-Prozess anfordern.

Daten, die in einer Transaktion an einen Prozess geschickt werden, werden dem Empfängerprozess vom Kerneltreiber in einem Transaktionspuffer bereitgestellt. Hierfür wird ein Teil des Kernelspeichers in den Adressraum des Userspace-Prozesses gemappt. Dieser Speicherbereich ist für den jeweiligen Userspace-Prozess nur lesbar, andere Prozesse ha-

ben keinen Zugriff auf diesen Speicherbereich. In der Transaktionsverarbeitung werden die Transaktionsinhalte vom Kerneltreiber direkt aus dem Adressraum des Absenderprozesses in den Transaktionspuffer des Empfängerprozesses kopiert.

In Transaktionen können Services, Binder-Proxies und Filedeskriptoren an andere Prozesse übergeben werden. Diese, als aktive Binder-Objekte, bezeichneten Objekttypen werden im Kerneltreiber bei der Transaktionsverarbeitung gesondert behandelt. Hierdurch wird bei der Übergabe eines Services an einen fremden Prozess, im Empfängerprozess ein Binder-Proxy instanziiert. Binder-Proxy Objekte können an dritte Prozesse übertragen werden. Diese Behandlung bildet die Grundlage für den Verbindungsaufbau zu einem Service über den ActivityManagerService.

Im Rahmen dieser Arbeit konnten verschiedene sicherheitsrelevante Aspekte des Binder-Frameworks betrachtet werden, hierbei konnten keine offensichtlichen Sicherheitslücken identifiziert werden. Vorbehaltlich der Sicherheit des Linux-Kernels, sind durch die Funktionsweise des Binder-Frameworks die Vertraulichkeit, Integrität und Authentizität von Transaktionen gewährleistet. Zur weiteren Verbesserung der Sicherheit, kann das SEAndroid-Framework genutzt werden, um die Kommunikation zwischen Prozessen mittels Access Control Listen einzuschränken.

Im Laufe der Untersuchungen zu dieser Arbeit stellte sich heraus, dass eine vollständige Betrachtung aller Aspekte und Eigenschaften des Binder-Frameworks den Rahmen dieser Arbeit übersteigt. Daher stehen nur die Transaktionsverarbeitung und die hierfür erforderlichen Grundlagen sowie deren Einbettung in den Gesamtkontext des Frameworks im Fokus dieser Arbeit. In den untersuchten Quelltexten finden sich an verschiedenen Stellen Hinweise auf Variationen in der Transaktionsverarbeitung. Diese sind in dieser Arbeit nicht berücksichtigt. Darüber hinaus werden im Binder-Framework Referenzen zwischen Prozessen verwaltet. Die Referenzverwaltung ist nicht Bestandteil dieser Arbeit. Da diese Funktionalität in der verfügbaren Literatur bisher kaum beschrieben ist, kann dies Gegenstand weiterführender Arbeiten sein.

In Kapitel 6 wird ein Teil der Funktionalität des ActivityManager-Service im Kontext des Verbindungsaufbaus zu einem Service beschrieben. Aufgrund des Umfangs des ActivityManagerServices, ist eine ausführliche Beschreibung dieses Services nicht Gegenstand dieser Arbeit. Es wird aber deutlich, dass der ActivityManagerService eine sicherheitskritische Komponente im Android-System darstellt, da er Verbindung zu allen Apps hat.

Neben der Funktionalität des Frameworks, verdeutlicht diese Arbeit die zentrale Rolle des Binder-Frameworks im Android-System. Aufgrund dieser Stellung bildet das Framework einen attraktiven Angriffspunkt für Angriffe. Gleichzeitig bietet es interessante Ansätze zur Verbesserung der Sicherheit des Betriebssystems und Anlass für weitere Untersuchungen in diese Richtung.

A. Anhang

A.1. Intents

Intents sind passive Datenstrukturen, die eine abstrakte Beschreibung einer auszuführenden Operation beinhalten. Intents dienen der Adressierung von Services. Intents ermöglichen zwei Formen der Adressierung:

- Explizit – Adressierung einer App-Komponente anhand ihres Namens
- Implizit – Adressierung über die auszuführende Operation (z.B. ACTION_VIEW content://contacts/people/)

Bei der impliziten Adressierung können mehrere Komponenten für die Ausführung der Operation in Frage kommen. In diesem Fall wird versucht, die Zielkomponente automatisch zu bestimmen. Ist dies nicht möglich, dann wird der Anwender zur Auswahl der Komponente aufgefordert.

Intents sind nur in Java implementiert, eine native Implementierung ist nicht vorhanden. Die Klasse Intent implementiert das Interface Parcelable und kann so in Transaktionen an andere Prozesse übertragen werden.

A.2. System-V-IPC

Semaphoren

Semaphoren diesen der Synchronisation von Prozessen beim Zugriff auf gemeinsam genutzte Ressourcen. Semaphoren sind Zähler, mit denen die Anzahl der konkurrierenden Zugriffe auf eine geschützte Ressource eingeschränkt wird. Bei jedem Zugriff auf eine Ressource wird der Zähler für die Dauer der Ressourcennutzung durch einen Prozess verringert. Nach Abschluss der Ressourcennutzung wird der Wert der Semaphore wieder erhöht. Steht der Zähler bei 0, dann sind keine weiteren Zugriffe möglich. Prozesse, welche die geschützte Ressource nutzen möchten, müssen warten, bis die Semaphore wieder freigegeben wird.

Shared Memory

Shared Memory sind Speichersegmente, welche in die virtuellen Adressräume mehrerer Prozesse gemappt werden und so von mehreren Prozessen gemeinsam genutzt werden können. Diese Form der IPC ist die schnellste Möglichkeit zum Datenaustausch zwischen Prozessen, da keine Strukturen oder Methoden zwischengeschaltet werden. Daten, welche in ein Shared Memory Segment geschrieben werden, stehen im gleichen Moment anderen Prozessen zur Verfügung. Üblicherweise wird der Zugriff auf Shared Memory-Segmente durch Semaphoren geschützt.

Vorbehaltlich der entsprechenden Berechtigungen können Shared Memory Segmente von mehreren Prozessen gleichzeitig genutzt werden.

Message Queues

Message Queues sind linked lists im Kernel Space, in denen Nachrichten gespeichert werden können, die dann von anderen Prozessen abgerufen werden. Nachrichten werden in der Reihenfolge in der Queue gespeichert, wie sie hinzugefügt werden. Jede Message Queue ist eindeutig durch ihren IPC-Identifier gekennzeichnet.

Message Queues werden mit dem Systemcall mssgget() erzeugt.

A.3. Sicherheitsziele

Sicherheitsziele beschreiben bestimmte Eigenschaften beziehungsweise Anforderungen, die ein System der Informationstechnik aufweisen oder erfüllen muss um die Sicherheit der Daten im System zu gewährleisten. Zu den am häufigsten genannten Sicherheitszielen gehören:

- Vertraulichkeit
- Integrität
- Authentizität

Vertraulichkeit

Die Vertraulichkeit beschreibt den Schutz von Daten vor dem Zugriff unbefugter Dritter. Die Vertraulichkeit kann beispielsweise durch Verschlüsselung von Daten oder der Nutzung vertrauenswürdiger Übertragungswege gewährleistet werden.

Integrität

Integrität beschreibt die Gewährleistung der Unversehrtheit bzw. Unverfälschtheit von Daten bei der Übertragung. Die Integrität einer Nachricht kann z.B. mittels einer Prüfsumme überprüft werden.

Authentizität

Authentizität beschreibt die Vertrauenswürdigkeit bzw. Echtheit einer Nachricht. Hierbei geht es vorrangig darum, dass bei einer Nachricht sichergestellt ist, dass gewährleistet wird, dass die Nachricht nicht von einem anderen als von dem in der Nachricht angegebenen Absender stammt. Dies kann beispielsweise anhand einer digitalen Signatur erfolgen.

A.4. Liste der Dateien

Im Rahmen der statischen Code-Analyse zu dieser Arbeit wurden die im Folgenden aufgelisteten Dateien einbezogen. Die Dateipfade sind relative Pfade ausgehend vom Checkout-Verzeichnis des Android Open Source Projektes bzw. des Linux-Kernels für Android.

Linux Kernel

- kernel/drivers/staging/android/uapi/binder.h
- kernel/drivers/staging/android/binder.h
- kernel/drivers/staging/android/binder.c

Android Open Source Projekt

- frameworks/native/cmds/servicemanager/binder.c
- frameworks/native/cmds/servicemanager/binder.h
- frameworks/native/cmds/servicemanager/service_manager.c
- frameworks/native/cmds/servicemanager/binder.c
- frameworks/native/libs/binder/IPCThreadState.cpp
- frameworks/native/include/binder/IPCThreadState.h
- frameworks/native/libs/binder/ProcessState.cpp
- frameworks/native/include/binder/ProcessState.h
- frameworks/native/libs/binder/Binder.cpp
- frameworks/native/include/binder/Binder.h
- frameworks/base/core/jni/android_util_Binder.cpp
- frameworks/base/core/jni/android_util_Binder.h
- frameworks/native/libs/binder/BpBinder.cpp
- frameworks/native/include/binder/BpBinder.h
- frameworks/native/libs/binder/Parcel.cpp

- frameworks/native/include/binder/Parcel.h
- frameworks/base/core/jni/android_os_Parcel.cpp
- frameworks/base/core/jni/android_os_Parcel.h
- frameworks/base/core/java/android/os/Binder.java
- frameworks/base/core/java/android/os/Parcel.java
- frameworks/base/core/jni/com_android_internal_os_ZygoteInit.cpp
- art/runtime/native/dalvik_system_Zygote.cc
- frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java
- frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
- frameworks/base/core/java/android/app/ActivityThread.java
- frameworks/base/core/java/android/app/ActivityManager.java
- frameworks/base/core/java/android/app/ActivityManagerNative.java
- frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

A.5. Binder Kommunikationsprotokoll

Das Binder-Kommunikationsprotokoll ist in der Datei `drivers/staging/android/uapi/binder.h` des Linux-Kernels für Android definiert. Die folgenden Code-Ausschnitte zeigen die Definitionen der Binder-Kommandos und Binder-Responses. In den Kommentaren zu den Kommandos sind jeweils die Parameter beschrieben, die in den Write- bzw. Read-Buffer geschrieben werden.

Binder Kommandos

```
1  enum binder_driver_command_protocol {
2  BC_TRANSACTION = _IOW('c', 0, struct binder_transaction_data),
3  BC_REPLY = _IOW('c', 1, struct binder_transaction_data),
4  /*
5   * binder_transaction_data: the sent command.
6   */
7
8  BC_ACQUIRE_RESULT = _IOW('c', 2, int),
9  /*
10  * not currently supported
11  * int:  0 if the last BR_ATTEMPT_ACQUIRE was not successful.
12  * Else you have acquired a primary reference on the object.
13  */
14
15 BC_FREE_BUFFER = _IOW('c', 3, int),
16 /*
17  * void *: ptr to transaction data received on a read
18  */
19
20 BC_INCREFS = _IOW('c', 4, int),
21 BC_ACQUIRE = _IOW('c', 5, int),
22 BC_RELEASE = _IOW('c', 6, int),
23 BC_DECREFS = _IOW('c', 7, int),
24 /*
25  * int:descriptor
26  */
27
28 BC_INCREFS_DONE = _IOW('c', 8, struct binder_ptr_cookie),
29 BC_ACQUIRE_DONE = _IOW('c', 9, struct binder_ptr_cookie),
30 /*
31  * void *: ptr to binder
32  * void *: cookie for binder
33  */
34
35 BC_ATTEMPT_ACQUIRE = _IOW('c', 10, struct binder_pri_desc),
36 /*
37  * not currently supported
38  * int: priority
39  * int: descriptor
40  */
41
42 BC_REGISTER_LOOPER = _IO('c', 11),
43 /*
44  * No parameters.
45  * Register a spawned looper thread with the device.
46  */
47
48 BC_ENTER_LOOPER = _IO('c', 12),
49 BC_EXIT_LOOPER = _IO('c', 13),
```

```
50 /*
51  * No parameters.
52  * These two commands are sent as an application-level thread
53  * enters and exits the binder loop, respectively.  They are
54  * used so the binder can have an accurate count of the number
55  * of looping threads it has available.
56  */
57
58 BC_REQUEST_DEATH_NOTIFICATION = _IOW('c', 14, struct binder_ptr_cookie),
59 /*
60  * void *: ptr to binder
61  * void *: cookie
62  */
63
64 BC_CLEAR_DEATH_NOTIFICATION = _IOW('c', 15, struct binder_ptr_cookie),
65 /*
66  * void *: ptr to binder
67  * void *: cookie
68  */
69
70 BC_DEAD_BINDER_DONE = _IOW('c', 16, void *),
71 /*
72  * void *: cookie
73  */
74 };
```

Binder-Responses

```
1  enum binder_driver_return_protocol {
2  BR_ERROR = _IOR('r', 0, int),
3  /*
4   * int: error code
5   */
6
7  BR_OK = _IO('r', 1),
8  /* No parameters! */
9
10 BR_TRANSACTION = _IOR('r', 2, struct binder_transaction_data),
11 BR_REPLY = _IOR('r', 3, struct binder_transaction_data),
12 /*
13  * binder_transaction_data: the received command.
14  */
15
16 BR_ACQUIRE_RESULT = _IOR('r', 4, int),
17 /*
18  * not currently supported
19  * int: 0 if the last bcATTEMPT_ACQUIRE was not successful.
20  * Else the remote object has acquired a primary reference.
21  */
22
23 BR_DEAD_REPLY = _IO('r', 5),
24 /*
25  * The target of the last transaction (either a bcTRANSACTION or
26  * a bcATTEMPT_ACQUIRE) is no longer with us.  No parameters.
27  */
28
29 BR_TRANSACTION_COMPLETE = _IO('r', 6),
30 /*
31  * No parameters... always refers to the last transaction requested
32  * (including replies).  Note that this will be sent even for
33  * asynchronous transactions.
34  */
35
36 BR_INCREFS = _IOR('r', 7, struct binder_ptr_cookie),
37 BR_ACQUIRE = _IOR('r', 8, struct binder_ptr_cookie),
38 BR_RELEASE = _IOR('r', 9, struct binder_ptr_cookie),
```

```
39 BR_DECREFS = _IOR('r', 10, struct binder_ptr_cookie),
40 /*
41  * void *:  ptr to binder
42  * void *: cookie for binder
43  */
44
45 BR_ATTEMPT_ACQUIRE = _IOR('r', 11, struct binder_pri_ptr_cookie),
46 /*
47  * not currently supported
48  * int:priority
49  * void *: ptr to binder
50  * void *: cookie for binder
51  */
52
53 BR_NOOP = _IO('r', 12),
54 /*
55  * No parameters.  Do nothing and examine the next command.  It exists
56  * primarily so that we can replace it with a BR_SPAWN_LOOPER command.
57  */
58
59 BR_SPAWN_LOOPER = _IO('r', 13),
60 /*
61  * No parameters.  The driver has determined that a process has no
62  * threads waiting to service incoming transactions.  When a process
63  * receives this command, it must spawn a new service thread and
64  * register it via bcENTER_LOOPER.
65  */
66
67 BR_FINISHED = _IO('r', 14),
68 /*
69  * not currently supported
70  * stop threadpool thread
71  */
72
73 BR_DEAD_BINDER = _IOR('r', 15, void *),
74 /*
75  * void *: cookie
76  */
77 BR_CLEAR_DEATH_NOTIFICATION_DONE = _IOR('r', 16, void *),
78 /*
79  * void *: cookie
80  */
81
82 BR_FAILED_REPLY = _IO('r', 17),
83 /*
84  * The the last transaction (either a bcTRANSACTION or
85  * a bcATTEMPT_ACQUIRE) failed (e.g. out of memory).  No parameters.
86  */
87 };
```

A.6. Datenstrukturen im Kerneltreiber

Die im Folgenden gezeigten Datenstrukturen werden im Kerneltreiber zur Abstraktion der Komponenten im Userspace-Prozess, zur Kommunikation mit dem Userspace-Prozess und in den internen Abläufen verwendet.

binder_proc

```
1  struct binder_proc {
2      struct hlist_node proc_node;
3      struct rb_root threads;
4      struct rb_root nodes;
5      struct rb_root refs_by_desc;
6      struct rb_root refs_by_node;
7      int pid;
8      struct vm_area_struct *vma;
9      struct mm_struct *vma_vm_mm;
10     struct task_struct *tsk;
11     struct files_struct *files;
12     struct hlist_node deferred_work_node;
13     int deferred_work;
14     void *buffer;
15     ptrdiff_t user_buffer_offset;
16     struct list_head buffers;
17     struct rb_root free_buffers;
18     struct rb_root allocated_buffers;
19     size_t free_async_space;
20     struct page **pages;
21     size_t buffer_size;
22     uint32_t buffer_free;
23     struct list_head todo;
24     wait_queue_head_t wait;
25     struct binder_stats stats;
26     struct list_head delivered_death;
27     int max_threads;
28     int requested_threads;
29     int requested_threads_started;
30     int ready_threads;
31     long default_priority;
32     struct dentry *debugfs_entry;
33 }
```

binder_thread

```
1  struct binder_thread {
2      struct binder_proc *proc;
3      struct rb_node rb_node;
4      int pid;
5      int looper;
6      struct binder_transaction *transaction_stack;
7      struct list_head todo;
8      uint32_t return_error; /* Write failed, return error code in read buf */
9      uint32_t return_error2; /* Write failed, return error code in read */
10     /* buffer. Used when sending a reply to a dead process that */
11     /* we are also waiting on */
12     wait_queue_head_t wait;
13     struct binder_stats stats;
14 }
```

binder_node

```
1  struct binder_node {
2      int debug_id;
3      struct binder_work work;
4      union {
5          struct rb_node rb_node;
6          struct hlist_node dead_node;
7      };
8      struct binder_proc *proc;
9      struct hlist_head refs;
10     int internal_strong_refs;
11     int local_weak_refs;
12     int local_strong_refs;
13     void __user *ptr;
14     void __user *cookie;
15     unsigned has_strong_ref:1;
16     unsigned pending_strong_ref:1;
17     unsigned has_weak_ref:1;
18     unsigned pending_weak_ref:1;
19     unsigned has_async_transaction:1;
20     unsigned accept_fds:1;
21     unsigned min_priority:8;
22     struct list_head async_todo;
23 };
```

binder_ref

```
1  struct binder_ref {
2      /* Lookups needed: */
3      /*   node + proc => ref (transaction) */
4      /*   desc + proc => ref (transaction, inc/dec ref) */
5      /*   node => refs + procs (proc exit) */
6      int debug_id;
7      struct rb_node rb_node_desc;
8      struct rb_node rb_node_node;
9      struct hlist_node node_entry;
10     struct binder_proc *proc;
11     struct binder_node *node;
12     uint32_t desc;
13     int strong;
14     int weak;
15     struct binder_ref_death *death;
16 };
```

binder_transaction_data

```
1  struct binder_transaction_data {
2   union {
3  /* WARNING: DO NOT EDIT, AUTO-GENERATED CODE - SEE TOP FOR INSTRUCTIONS */
4   size_t handle;
5   void *ptr;
6   } target;
7   void *cookie;
8  /* WARNING: DO NOT EDIT, AUTO-GENERATED CODE - SEE TOP FOR INSTRUCTIONS */
9   unsigned int code;
10  unsigned int flags;
11  pid_t sender_pid;
```

```
12  uid_t sender_euid;
13 /* WARNING: DO NOT EDIT, AUTO-GENERATED CODE - SEE TOP FOR INSTRUCTIONS */
14  size_t data_size;
15  size_t offsets_size;
16  union {
17  struct {
18 /* WARNING: DO NOT EDIT, AUTO-GENERATED CODE - SEE TOP FOR INSTRUCTIONS */
19  const void *buffer;
20  const void *offsets;
21  } ptr;
22  uint8_t buf[8];
23 /* WARNING: DO NOT EDIT, AUTO-GENERATED CODE - SEE TOP FOR INSTRUCTIONS */
24  } data;
25 };
```

binder_transaction

```
1  struct binder_transaction {
2  int debug_id;
3  struct binder_work work;
4  struct binder_thread *from;
5  struct binder_transaction *from_parent;
6  struct binder_proc *to_proc;
7  struct binder_thread *to_thread;
8  struct binder_transaction *to_parent;
9  unsigned need_reply:1;
10 /* unsigned is_dead:1; */   /* not used at the moment */
11
12 struct binder_buffer *buffer;
13 unsigned int code;
14 unsigned int flags;
15 long   priority;
16 long   saved_priority;
17 kuid_t sender_euid;
18 };
```

binder_buffer

```
1  struct binder_buffer {
2  struct list_head entry; /* free and allocated entries by address */
3  struct rb_node rb_node; /* free entry by size or allocated entry */
4              /* by address */
5  unsigned free:1;
6  unsigned allow_user_free:1;
7  unsigned async_transaction:1;
8  unsigned debug_id:29;
9
10 struct binder_transaction *transaction;
11
12 struct binder_node *target_node;
13 size_t data_size;
14 size_t offsets_size;
15 uint8_t data[0];
16 };
```

binder_work

```
1  struct binder_work {
```

```
2  struct list_head entry;
3  enum {
4    BINDER_WORK_TRANSACTION = 1,
5    BINDER_WORK_TRANSACTION_COMPLETE,
6    BINDER_WORK_NODE,
7    BINDER_WORK_DEAD_BINDER,
8    BINDER_WORK_DEAD_BINDER_AND_CLEAR,
9    BINDER_WORK_CLEAR_DEATH_NOTIFICATION,
10 } type;
11 };
```

binder_write_read

```
1  struct binder_write_read {
2  signed long write_size;     /* bytes to write */
3  signed long write_consumed; /* bytes consumed by driver */
4  unsigned long    write_buffer;
5  signed long read_size; /* bytes to read */
6  signed long read_consumed;  /* bytes consumed by driver */
7  unsigned long    read_buffer;
8  };
```

Literaturverzeichnis

Android Open Source Project. (2014). *Implementing Security*. Abgerufen am 08. 12 2014 von Android Open Source Project: https://source.android.com/devices/tech/security/implement.html

Android Open Source Project. (2013). *ndk/docs/text/system/libc/SYSV-IPC.text*. Abgerufen am 23. 10 2014 von github: https://github.com/awong-dev/ndk/blob/master/docs/text/system/libc/SYSV-IPC.text

Artenstein, N., & Reviso, I. (2014). *Man in the Binder: Who controls IPC controls the Droid*. Report, Check Point Software Technologies Ltd., Malware Research Lab.

Brady, P. (2008). Android Anantomy and Physiology. *Google I/O*.

Chin, E., Porter Felt, A., Greenwood, K., & Wagner, D. (2011). *Analyzing Inter-Application Communication in Android*. Paper, University of California, Berkeley.

Corbet, J., Rubini, A., & Kroah-Hartman, G. (2005). *Linux Device Drivers* (3. Auflage Ausg.). O'Reilly & Associates.

Devos, M. (2013). *Bionic vs. GlibC Report*. Master Thesis, Universität Gent.

Hausner, C. (2014). *Binderwall: Monitoring an Filtering Android Interprocess Communication*. Master Thesis, Technische Universität München, Fakultät für Informatik.

Hellmann, E. (2013). *Android programming: Pushing the limits* (1. Auflage Ausg.). John Wiley & Sons.

Lockwood, A. (2013). *Android Design Patterns*. Abgerufen am 15. 02 2015 von http://www.androiddesignpatterns.com/2013/07/binders-window-tokens.html zu finden.

Pazdera, R. (10. 11 2012). *broken.build*. Abgerufen am 02. 03 2015 von http://broken.build/2012/11/10/magical-container_of-macro/

Rivera, J., & van der Meulen, R. (15. 12 2014). *Gartner Says Sales of Smartphones Grew 20 Percent in Third Quarter of 2014*. (Gartner Inc.) Abgerufen am 30. 04 2015 von Gartner Inc Website: http://www.gartner.com/newsroom/id/2944819

Rosa, T. (2011). *Android Binder Security Note: On Passing Binder Through Another Binder*.

Rusling, D. (1999). *The Linux Documentation Project*. Abgerufen am 03. 12 2014 von http://www.tldp.org/LDP/tlk/ds/ds.html

Schreiber, T. (2011). *Android Binder - Android Interprocess Communication*. Seminar Thesis, Ruhr-Universität Bochum.

Glossar

Application Programming Interface (API)

Ein Application Programming Interface ist eine Programmierschnittstelle, welche die Nutzung eines Systems aus fremden Anwendungen ermöglicht. APIs können beispielsweise in Form von Softwarebibliotheken bereitgestellt werden.

Access Control List (ACL)

Access Control Listen (Zugriffskontrolllisten) ermöglichen die Definition von Zugriffsbeschränkung für Prozesse oder Benutzer in Systemen der Informationstechnik anhand von Regelwerken.

C-Library

Die C-Library ist eine Bibliothek für die Programmiersprache C. Diese Bibliothek stellt Standardfunktionen und Funktionen für die Nutzung von Systemzugriffen zur Verfügung.

Copy-On-Write

Das Copy-on-Write Verfahren dient zur Reduzierung des Hauptspeicherverbrauchs in Betriebssystemen. Hierbei werden Daten im Hauptspeicher eines Rechners bei gleichzeitiger Nutzung durch mehrere Prozesse erst dann in einen neuen Speicherbereich kopiert, wenn diese durch einen Prozess verändert werden.

Debugger

Ein Debugger ermöglicht die Analyse von Software-Anwendungen zur Laufzeit. Durch Debugging ist es möglich, einen Prozess zur Laufzeit zu unterbrechen, in Einzelschritten fortzusetzen und so den inneren Zustand der Anwendung während der Ausführung zu untersuchen.

Dbus

Dbus ist ein IPC_Framework für Linux, das die Nachrichtenorientiere Kommunikation zwischen Prozessen ermöglicht. Dbus orientiert sich vorrangig an den Bedürfnissen von Desktop-Anwendungen.

General Public License

Die General Public License ist eine von der Free Software Foundation veröffentlichte Softwarelizenz. Die GPL schreibt vor, dass Änderungen oder Ableitungen von einem Werk unter der GPL ebenfalls unter der GPL veröffentlicht werden müssen.

Java Native Interface (JNI)

Das Java native Interface ermöglicht die Nutzung nativer in C++ entwickelter Anwendungskomponenten aus Java heraus.

Lesser General Public License

Die Lesser General Public License ist eine von der GPL abgeleitete Lizenz. Im Gegensatz zur GPL erlaubt die LPGL die Einbindung von Softwarekomponenten, die unter LGPL stehen in Werke, die unter einer anderen Lizenz stehen.

Sandboxing

Sandboxing ist eine Technik zur Isolation von Programmen in einem Betriebssystem. Anwendungen in einer Sandbox haben nur Zugriff auf Daten innerhalb der Sandbox und können nicht auf Daten des Betriebssystem oder anderer Anwendungen zugreifen.

Software Development Kit (SDK)

Ein Software-Development-Kit ist eine Sammlung von Werkzeugen zur Softwareentwicklung. Ein SDK kann z.B. Softwarebibliotheken, Compiler, Debugger und eine Entwicklungsumgebung umfassen.

Stub

Der Begriff Stub bezeichnet in der Softwareentwicklung eine Komponente (z.B. eine Methode oder eine Klasse), die nur rudimentäre Funktionalität umfasst und stellvertretend für eine funktionale Implementierung steht.

Tabellenverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings