RUB

RUHR-UNIVERSITÄT BOCHUM

Android Binder

Android Interprocess Communication

Thorsten Schreiber
First Advisor: Juraj Somorovsky
Second Advisor: Daniel Bußmeyer

Seminarthesis
October 5, 2011
Chair for Network and Data Security: Prof. Dr.-Ing. Jörg Schwenk

Abstract

This paper is an analysis of the interprocess communication in Android mobile operating system, provided by a custom software called Binder. It gives an overview about the capabilities and different layers of the Binder framework.

Contents

1. Introduction

A Gartner forecast [8] stated in late 2010, that Android will "... challenge symbian for no. 1 position by 2014".That means, that Android is an increasing factor in smartphone computing. Android introduces new extensions and features to the Linux kernel. The Binder framework for interprocess communication represents such a feature. For this framework a complete documentation does not exist. The Java classes are well documented, but it gets more fragmented and is completely missing for the kernel module.

This work tries to give an overview of the Binder framework, its features and functionalities.

Chapter 2 provides all needed background knowledge to understand the Binder framework. It gives an overview about Linux and outlines its basic concepts.

Chapter 3 introduces the Android operating system and its basic concepts.

Chapter 4 discusses the Binder and its capabilities. The explanation covers an abstract level, without going into implementation details.

The Chapter 5 discusses the different layers of the Binder framework and where its the features are implemented.

Chapter 6 gives an example of an applied Binder interprocess communication, where an Android client application accesses an Android remote service application and performs an remote procedure call.

2. Background

In this chapter we discuss the underlying theory. We give an overview on the topic of multitasking and processes and we point out why the concept of interprocess communication is needed.

2.1. Multitasking, Processes and Threads

Multitasking is the ability to execute multiple instances of programs or processes at the same time. An *operating system* therefore creates for every binary executable file a certain memory frame with its own stack, heap, data and shared mapped libraries. It also assigns special internal management structures. This is called a *process*.

The operating system must provide fair proportioning, because only one process can use the CPU at the same time. All processes must be interruptible. The operating system sends them to sleep or wakes them on their time slot. This work is done by a *scheduler*, supplying each process with an optimal time slot.

A *thread* is a process without own address space in memory, it shares the address space with the parent process. Processes can have child threads, and a thread must be assigned to a process.

2.2. Process Isolation

Due to security and safety reasons, one process must not manipulate the data of another process. For this purpose an operating system must integrate a concept for *process isolation*. In Linux, the *virtual memory mechanism* achieves that by assigning each process accesses to one linear and contiguous memory space. This virtual memory space is mapped to physical memory by the operating system. Each process has its own virtual memory space, so that a process cannot manipulate the memory space of another process. The memory access of a process is limited to its virtual memory. Only the operating system has access to physical and therefore all memory.

The process isolation ensures for each process memory security, but in many cases the communication between process is wanted and needed. The operating system must provide mechanisms to approve interprocess communication.

2.3. User Space and Kernel Space

Processes run normally in an unprivileged operation mode, that means they have no access to physical memory or devices. This operation mode is called in Linux *user space*. More abstractly, the concept of security boundaries of an operating system introduces the term *ring*. Note, that this must be a hardware supported feature of the platform. A certain group of rights is assigned to a ring. Intel hardware [21] supports four rings, but only two rings are used by Linux. These are ring 0 with full rights and ring 3 with least rights. Ring 1 and 2 are unused. System processes run in ring 0 and user processes in ring 3. If a process needs higher privileges, it must perform a transition from ring 3 to ring 0. The transition passes a gateway, that performs security checks on arguments. This transition is called *system call* and produces a certain amount of calculating overhead.

2.4. Interprocess Communication in Linux

If one process exchanges data with another process, it is called *interprocess communication* (IPC). Linux offers a variety of mechanisms for IPC. These are the following listed: [23]

Signals Oldest IPC method. A process can send signals to processes with the same uid and gid or in the same process group.

Pipes Pipes are unidirectional bytestreams that connect the standard output from one process with the standard input of another process.

Sockets A socket is an endpoint of bidirectional communication. Two processes can communicate with bytestreams by opening the same socket.

Message queues Processes can write a message to a message queue that is readable for other Processes.

Semaphores A semaphore is a shared variable that can be read and written by many processes.

Shared Memory A location in system memory mapped into virtual address spaces of two processes, that each process can fully access.

3. Android

In this chapter, the basic mechanisms and concepts of the mobile operating system Android are presented. Android was developed by the Open Handset Alliance and Google and is available since 2008. [1]

3.1. Kernel

Android is based on a Linux 2.6 standard kernel but enhanced with new extensions for mobile needs. These are kernel modules *Alarm*, *Ashmem*, *Binder*, power management, *Low Memory Killer*, a kernel debugger and a logger. We will analyze the Binder driver in this work, that offers a new IPC mechanism to Linux. [17]

3.2. Programming Languages

Four programming languages are used for system development: Assembler, C, C++ and Java. The kernel has a small amount of Assembler but is mainly written in C. Some native applications and libraries are written in C++. All other applications, especially custom apps, are written in Java. [10]

3.3. Java Native Interface

A distinction is made between programs compiled for the virtual machine and programs compiled to run on a specific computation platform, like Intel x86 or ARM. Programs compiled for a specific platform are called native. Because Java is executed in a virtual machine with its own byte-code, no native code can be executed directly. Due to the need to access low-level os mechanism like kernel calls, Java has to overcome this obstacle. This is done by the *Java native interface* (JNI) [22], which allows Java to execute compiled code from libraries written in other languages, e.g. C++. This is a trade-off between gaining capabilities of accessing the system and decreasing the level of security in Java.

3.4. Dalvik Virtual Machine

The *Dalvik virtual machine* (DVM) [5] runs the Java programmed apps. The DVM does not claim to be a Java virtual machine (JVM) due to license reasons, but fulfills the same purpose. Java 5 programs can run in that environment.

The Sun JVM is stack based, because a stack machine can be run on every hardware. Hardware and platform independence were major design principles of Java. The DVM is register based for performance reasons and well adapted to ARM hardware. This is a different design principle, taking the advantage of hardware independence for high performance and less power consumption, which is essential for mobile purposes with limited battery capability. The possibility to use the Java native interface weakens the security guarantying property of Java to implicit checking the bounds of variables and to encapsulate system calls and the force to use JVM defined interfaces to the system. The use of native libraries can allow bypassing the type and border checking of the virtual machine and opens the door to stack-overflow attacks. [4]

Even it is a security issue, the JNI is essential for the interprocess communication mechanism because the middleware of Binder are C++ libraries and must be accessed with JNI.

3.5. Zygote

Due to performance reasons, the DVM is started only once. Each new instance of it is cloned. This is done by a system service called *Zygote*. [2]

First, it preinitializes and preloads common Android classes in its heap. [25] Then, it listens on a socket for commands to start a new Android application. On receiving a start command, it forks a new process with the loaded application. This process becomes the started application and shares the heap with the original Zygote process by copy-on-write mapping and so the memory pages of Zygote's heap are linked to this new process. While the application reads only from the heap, it stays shared. But when the application performs write operations on its heap, the corresponding memory page is copied and the link is changed to the new page. Now the heap can be manipulated, without manipulating the original data from the parent Zygote process.

When an Android application forks, it uses Zygote's memory layout and therefore the layout is the same for each application.

3.6. Application Concept

Each Android application is composed from up to 4 different components. [12] Each component has a special subject. Figure 3.1 presents the components as a hierarchically class diagram since they are actually Java classes.

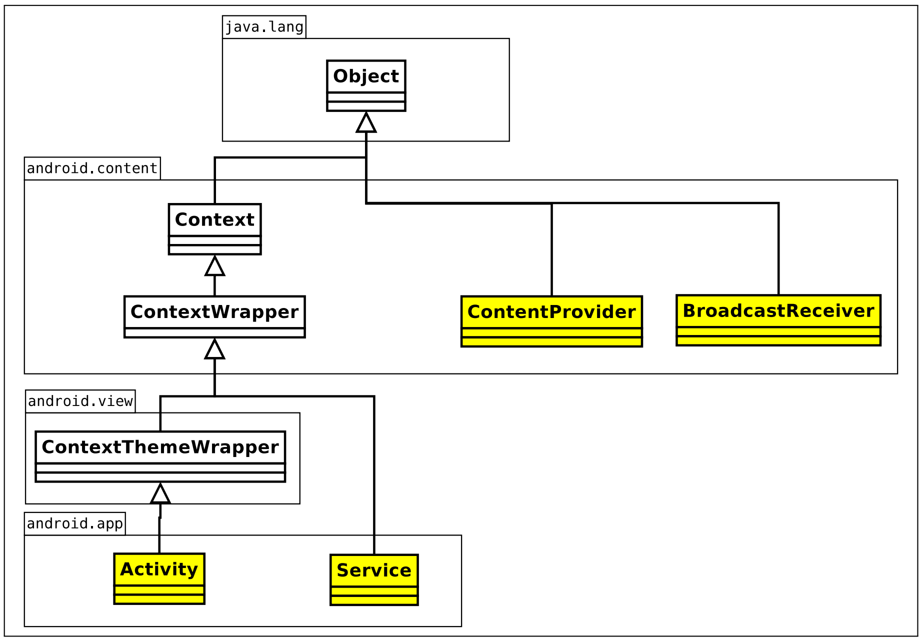


Figure 3.1.: Application Components System

The *activity* represents the user interface of an application. It is responsible for performing the screen and receiving interaction created by the user. It is not intended to hold persistent data because it can be sent to sleep by the operating system if another activity is brought to the front.

For long duration purposes Android offers the *service* component. All tasks running in the background of an application must be implemented here, because a foreground service is only stopped if the system runs out of memory and apps must be terminated to free memory.

Even if the service is persistent in task performing, it is depreciated to hold persistent data. This is the subject of the *content provider*, which gives an interface for accessing persistent data like file or network streams as SQL-like databases.

The *broadcast receiver* is for receiving system wide messages, i.e. the message that a new SMS has come in is provided to all subscribers. A low battery level warning is also sent on this channel. *Broadcast reveivers* handle these messages and marshal certain action, e.g. saving the state of an app in prospect to a soon shutdown of the mobile device.

The *application manifest* [13] keeps the information for Android about the component. In this file, the basic application configuration is set. E.g., if an service starts in its own process or if it is attached to local process. Listing 3.1 gives an example of an Android application manifest XML file.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.binder">
    <application android:icon="@drawable/icon">
        <activity android:name=".ActivityExample" android:label="
            @string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
                    LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".SimpleMathService"
           android:process=":remote">
            <intent-filter android:priority="25">
              <action android:name="com.msi.manning.binder.
                  ISimpleMathService"></action>
            </intent-filter>
            </service>
    </application>
</manifest>
```

Listing 3.1: Example Manifest

3.7. Component Communication Concepts

As different components have to exchange data, this is realized through intercomponent communication, or interprocess communication, if the specific components belong to different processes (apps).

The communication works with so called intents. These are representations for operations to be performed. An intent is basically a datastructure which contains a `URI` and an `action`. The `URI` uniquely identifies an application component and the `action` identifies the operation to be executed.

This intent is submitted by the interprocess communication system. Figure 3.2 gives an overview about the different forms of component interaction. An activity is started by an intent and can bring another activity to the front.

A service can be started, stopped and bound by IPC. Also the call and return methods are implemented by IPC.

A content provider can be queried by an activity via IPC and returns the result accordingly. The Android source code files show an extensive use of IPC to exchange the abstract data.

A broadcast receiver gets all the intents (eg. messages) it has subscripted to via IPC. [12]

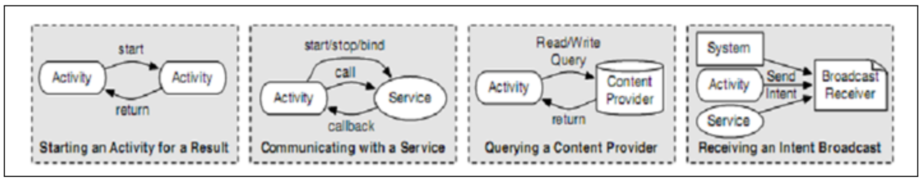


Figure 3.2.: Application Components System[18]

At this point, the importance of the IPC mechanism becomes apparent. The Android OS with its framework is a distributed system and the major and key technology to achieve that design is the IPC Binder mechanism.

3.8. Security Concept

The security mechanism in Android consists of three layers. The basic layer consists of a division of the persistent memory in two partitions, called system and data. The system partition is mounted as read only to prevent system data manipulation. The data partition is the place where application states and persistent data can be stored. Note, that the system partition can be remounted in write mode by the *App Store* application to install new apps.

To separate the apps from each other, the discretionary access control (DAC) model of Linux is used. Each app has its unique user ID and group ID and can only access its own directory. Only apps from the same author run under the same user ID and can access their data. All apps must be signed by the author to prevent data manipulation and to identify the author. So the system can determine if apps are from the same author and can run under one UID.

Since the Linux DAC model allows only a rudimentary level of access control, for fine granulated rights Android's middleware offers a permission label system which implements a mandatory access control (MAC) model. This system is based on a set of permissions and a set of exported services including access restrictions.

For each action performed on the operating system a permission label exists. At installation time the application asks the user for a set of permissions, the user has the choice between granting all asked permissions or aborting the installation. Once granted at installation time, a permission can never be removed except by uninstalling the app.

Every app can specify an intent filter, a white list mechanism, that defines the types of intents the components of the application should receive. Intents that are not listed will be filtered out by the reference monitor.

A problem is that the apps of an author can communicate freely because of same UID and GID. That means, if multiple apps from the same author are installed on the phone the different rights of each app accumulate through transitive property. App A asks app B of the same author to perform an action, for which app A has no rights but app B. So app B gets in possession of rights that are not granted by user for this dedicated app. [6] [18] [16]

4. Binder

This chapter explains what the Binder is and what its capabilities are. The explanation covers an abstract level, without going into implementation details. These are handled in the next chapter.

4.1. Origin

The Binder was originally developed under the name OpenBinder by Be Inc and later Palm Inc under the leadership of Dianne Hackborn. Its documentation claims OpenBinder as ”... a system-level component architecture, designed to provide a richer high-level abstraction on top of traditional modern operating system services.”

Or more concretely, the Binder has the facility to provide bindings to functions and data from one execution environment to another. The OpenBinder implementation runs under Linux and extends the existing IPC mechanisms. The documentation of OpenBinder states that ”... the code has run on a diverse variety of platforms, including BeOS, Windows, and PalmOS Cobalt.” [20]

Binder in Android is a customized implementation of OpenBinder. In Android's M3 release the original kernel driver part from OpenBinder was used, but the user space part had to be rewritten due to the license of OpenBinder beeing not compatible to Android's license. In Android's M5 release the driver was rewritten, too. The core concepts remained the same, but many details have changed.

The original OpenBinder code is no longer being developed. But to some extent, the reimplementation to android can be seen as a branch, which is maintained by Android developers. So the OpenBinder survived in Android Binder. [19]

4.2. Binder Terminology

The Binder framework uses its own terminology to name facilities and components. This section summarizes the most important terms, the details are discussed in the later sections. [20]

Binder This term is used ambiguous. *The* Binder refers to the overall Binder architecture, whereas *a* Binder refers to a particular implementation of a Binder interface.

Binder Object is an instance of a class that implements the Binder interface. A Binder object can implement multiple Binders.

Binder Protocol The Binder middleware uses a very low level protocol to communicate with the driver.

IBinder Interface A Binder interface is a well-defined set of methods, properties and events that a Binder can implement. It is usually described by AIDL[1] language.

Binder Token A numeric value that uniquely identifies a Binder.

4.3. Facilities

The Binder framework provides more than a simple interprocess messaging system. The facilities are listed in Figure 4.1

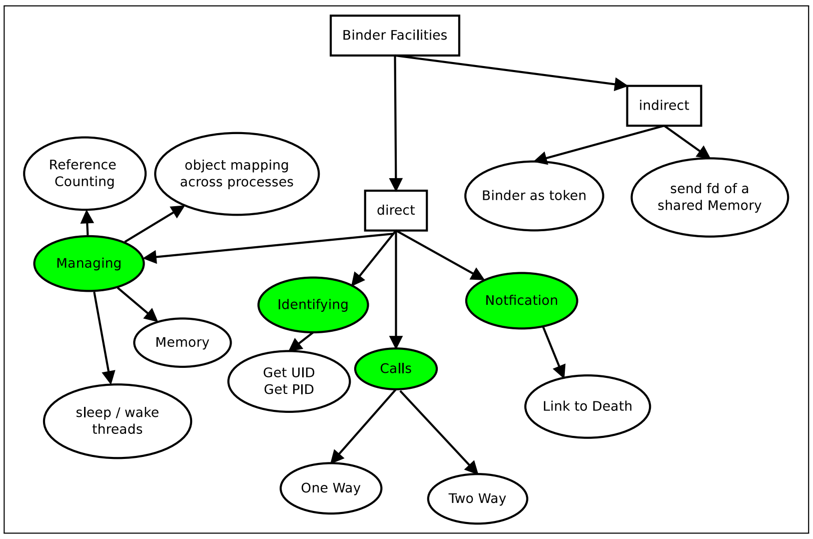


Figure 4.1.: The Binder Facilities

The most important improvement from Android application programmer's view is that methods on remote objects can be called as if they where local object methods. This is achieved with a synchronous method call. Accordingly, the calling client process is blocked for the duration of the answer of the server

[1]see Section 5.1

process. To its advantage, the client has no need to provide a threat method for a asynchronous return message from the client.

This is related to the facility of the Binder framework to send one and two way messages and to start and stop threads.

It is a feature of AIDL, thus settled in a higher level, that an application does not need to know if a service is running in a server process or in the local process. Android's application concept makes it possible to run a service either in a own process or in the activity process. This makes it easy for a application developer to export services to other Android applications without reviewing the code.

The Android *system service* uses a special notification feature of the Binder, that is called *link to death* mechanism. This facility allows processes to get informed when a Binder of a certain process is terminated. In particular, this is the way the Android *window manager* establishes a *link to death* relation to the callback Binder interface of each window, to get informed if the window is closed.

Each Binder is uniquely identifiable, that means it can act as shared token. Under the assumption the Binder is not published via the *service manager*, the Binder identification is only known by the involved communicating parties, including remote process, local process and the system. Consequently a Binder may be used as a security access token. The token can also be shared across multiple processes.

Another security feature is that a callee process can identify his caller process by UID and PID. Combined with the Android security model, a process can be identified. Another feature, but in this work not analyzed, is the shared memory mechanism, where via binder framework a heap can be shared.

Summarized, the Binder and its framework support many features to ensure a well object oriented interprocess communication. [19] [14]

4.4. Concepts

This section describes the concepts behind the facilities. Due to the lack of a public available documentation, they are largely derived by reviewing the source code.

4.4.1. Communication Model

The Binder framework communication is a client server model. A client will initiate a communication and wait for response from a server. The Binder framework uses a client-side proxy for communication. On the server side, a thread pool exists for working on requests. In Figure 4.2 the process A is the client and holds the proxy object which implements the communication with the Binder kernel driver. Process B is the server process, with multiple Binder threads. The Binder framework will spawn new threads to handle all incoming requests, until a defined maximum count of threads is reached. The proxy objects are talking to the Binder driver, that will deliver the message to the destinated object. [3]

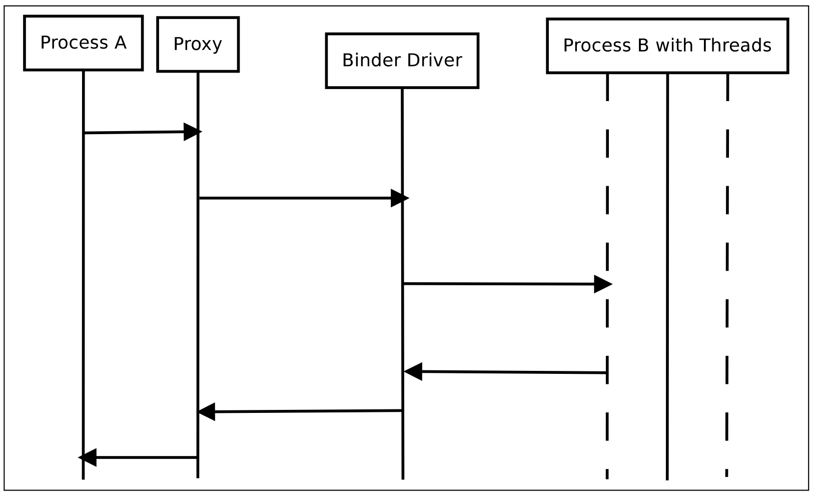


Figure 4.2.: Abstract Binder Communication

4.4.2. Transactions

If one process sends data to another process, it is called transaction. Along with each transmission payload data is submitted. The data is called transaction data.

The structure of the data is shown in Figure 4.3. It contains a target, that is a destination binder node. The `cookie` field is used for internal information. The `sender ID` field contains security relevant information. The `data` field contains a serialized data array. An array entry is composed of a command and its arguments, that is parsed through the Binder. An application on top of the Binder framework can now define own commands and depending on that arguments. As it can be seen in Android source codes derived from AIDL [11], developers implementing remote services use the `target command` field as a function pointer and serialize its arguments to the argument field corresponding to the function signature. Beyond this fact, the concept facilitate to implement a user defined IPC protocol. Binder communication is relying on a request and reply mechanism. This fact limits the implementation, meaning that each request must have a reply and the requesting process must wait for it. It is blocked in waiting time and cannot proceed. To implement an asynchronous IPC protocol, which is possible in general, a developer must pay attention to this fact. If a developer needs a non-blocking IPC, he has to implement a managing mechanism that assigns the answer to its request. Binder does not provide this asynchronous feature.

`Target`	`Binder Driver Command`	`Cookie`	`Sender ID`	`Data:`	
				`Target Command` 0	Arguments 0
				Target Command 1	Arguments 1
				...	...
				Target Command n-1	Arguments n-1

Figure 4.3.: Transmission Data

Therefore a transaction implies two messages, a transaction request and its reply. The one way communication of the Binder facilities section is limited to internal features as the *death notification*.

4.4.3. Parcels and Marshaling

In an object oriented view, the transaction data is called `parcel`. Namely, this is the transmitted data structure. Any object, that can be transmitted remotely, must implement the `Parcelable` interface. An object, that implements this interface must provide methods, that serialize the data structure of an object on sender side and restore it on receiver side. All information must be reduced to simple data types like `Int`, `Float`, `Boolean` and `String` types. This partial information are written serial to an parcel by the sender and are read and rebuild by the receiver.

The procedure of building a parcel is called marshaling or flattening an object. In reverse, the procedure of rebuilding a object from a parcel is called unmarshaling or unflattening an object.

The memory sharing facilities of Binder can not be used by Java API wrapper applications, only native C++ libraries can have access to shared object representations in memory.

4.4.4. Death Notification

The Binder framework supports notifications on the death of Binder objects. This is realized by an observer pattern. A local Binder object, that is interested to know about the termination of a remote Binder object adds itself to an observer list. If the event occurs that a process keeping the remote Binder object is terminated, the local object is informed and can react. Figure 4.4 was discharged from source code and is a flow chart and visualizes the pattern. [19]

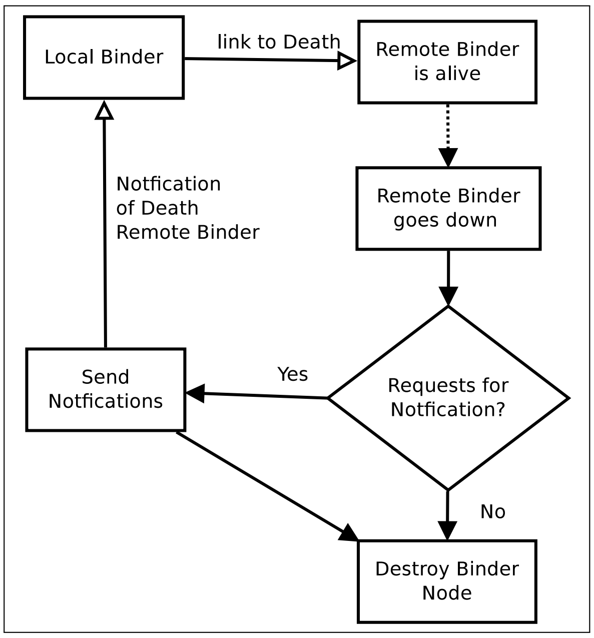


Figure 4.4.: Binder Death Notification

4.5. Context Manager

The *context manager* is a special Binder node of the Binder framework. It is the Binder with the number 0. It serves as name system and makes it possible to assign a name to a Binder interface. This is important, because possible clients do not know the remote Binder addresses a priori. If the client would know the remote Binder address a priori, the security token feature of Binder would not work. But if no remote partner address is known, no initial communication could happen, because each Binder interface knows only his own address. The context manager solves this problem, having the only Binder interface with a fixed and a priori known Binder address. The *context manager* implementation is not part of the Binder framework. In Android the implementation of the context manager is called *service manager*. Each Binder that needs to publish its name due to being a service, submits a name and its Binder token to the service manager. Relying on that feature, the client must only know the name of a service and asks the service manager for the Binder address of the requested service.

4.6. Intents

An *intent* is a message, that the developer uses on Java API layer and that is sent with Binder IPC. "It is an abstract representation of an operation to be performed." [15] "Abstract" means that the performer of the desired operation does not have to be defined in the intent. The intent holds as main information an `action` and a `data` field. Figure 4.5 gives an example of an intent. This intent is delivered by the intent reference monitor to the Binder that is assigned to the action ACTION_DIAL, e.g. the telephony application. This service will dial the number, that is stored in the contact application under the given name.

Action = ACTION_DIAL	Data = content://contacts/people/1

Figure 4.5.: Intent

There are two forms of intents. An explicit intent addresses to a specific component. On the other side, an implicit intent gives the decision to the Android system, which component is addressed. If multiple components for one purposes are installed, the system will choose the best component to run the intent.

4.7. System Integration

The Binder is extensively used in the android platform. Whenever two processes must communicate, Binder is involved. For example the window-manger exchange data with his clients over Binder framework. It also uses the Binder death notification feature for getting informed when a client application terminates.

So the Binder connects the distributed architecture of the Android operating system and is because of this a really important module. [19]

4.8. Security Model

The Binder security model is very basic but effective. It ensures a secure channel for communication of two process, and guarantees identification of communication partners by delivering information like PID number and UID number.

Another feature coming with intents is the intent filter. This is a declaration for an service or app, which intents are forwarded by the system to this service or app. But it does not guarantee security for all intents, because the intent filter can be bypassed by explicit intents. [24] In last consequence, the security relies on checking of PID and UID like presented above.

5. Implementation of the Binder Framework

This chapter provides an overview about the implementation of the Binder framework. For each layer, the source code files [10] [9] are listed and its purposes are discussed. Also, the AIDL is presented, which is an implementation feature because it generates Java code and thus can be viewed as a part of the Binder framework.

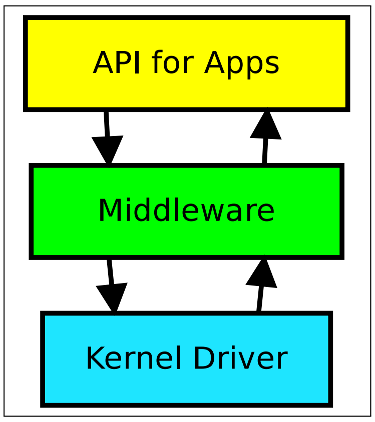


Figure 5.1.: IPC System

Figure 5.1 presents the different layers of the Binder framework. It is composed of three layers. The first and highest layer is the API for Android applications. The second layer is a middleware that keeps the userspace implementation of the Binder framework. The third and lowest layer is the kernel driver.

5.1. AIDL

The *Android interface definition language* (AIDL) [11] is part of the Eclipse SDK, provided by Google. Its main purpose is to ease the implementation of Android remote services. The AIDL follows a Java like syntax.

In the AIDL file the developer defines an interface with the method signatures of the remote service. The AIDL parser generates a Java class from the interface, that can be used for two different purposes. First it generates a proxy class to give the client access to the service, second it generates a stub class that can be used by the service implementation to extend it to an anonymous class with the implementation of the remote methods.

The AIDL language supports only basic data types. It generates code that takes care of writing the values into the parcels, sending them via Binder IPC, receiving them, reading the values and calling the methods of service and writing and sending the result back.

The AIDL file must be shared between remote service app developer and client app developer. Because the AIDL generator generates the source code for client and remote service in one file, each application uses and instantiate only a subset of generated classes.

5.2. Java API Wrapper

This section discusses the Java framework, operating on top of the middleware and the kernel driver. These source classes and interfaces belong to the Java API layer:

Interfaces:

- `android.app.IActivityManager`
- `android.os.Parcable`
- `andorid.os.IBinder`
- `android.content.ServiceConnection`

Classes:

- `android.app.ActivityManagerNative`
- `android.app.ContextImpl`
- `android.content.Intent`
- `android.content.ComponentName`
- `android.os.Parcel`
- `android.os.Bundle`
- `android.os.Binder`
- `android.os.BinderProxy`
- `com.android.internal.os.BinderInternal`

The Java layer of the Binder framework has two functions. One function is wrapping the subjacent middleware layer, to let the Android applications participate on Binder communication. As a second function, it introduces facilities to the Binder framework, namely the use of intents.

Figure 5.2 presents the main Java classes and its dependencies.

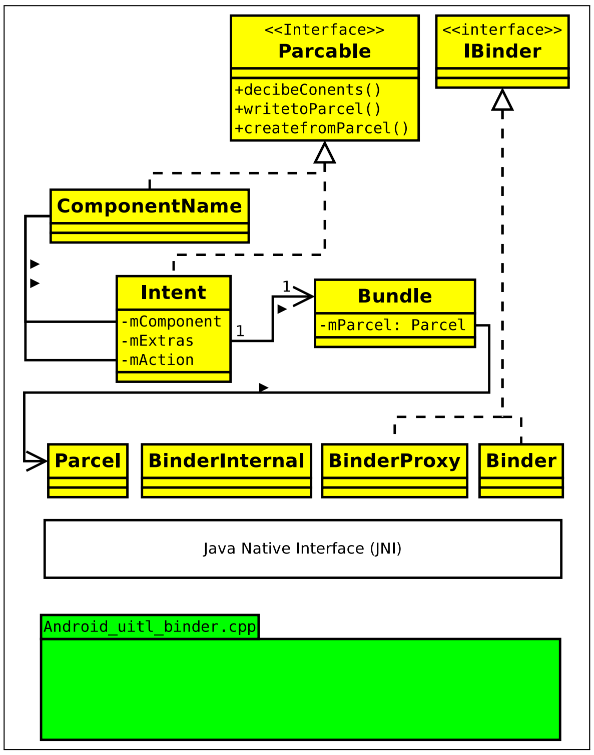


Figure 5.2.: Java System

5.2.1. JNI Wrapper

The Java API Layer relies on the Binder middleware. To use the C++ written middleware from Java, the JNI must be used. In the source code file `frameworks/base/core/jni/android_util_Binder.cpp`, the mapping between Java and C++ function is realized.

5.3. C++ Middleware

The middleware implements the user space facilities of Binder framework and is written in C++. The framework provides the process and thread control methods and structures that are necessary to spawn and manage new threads for working on requests. The marshalling and unmarshalling facilities are implemented here, so that the object information can be transformed to a submittable parcel of data. The middleware provides the interaction with the Binder kernel driver and implements the shared memory.

Services or apps written in native C++ can use the Binder framework directly, but must relinquish features implemented in Java API layer.

The source code is contained in these files:

- `frameworks/base/include/utils/IInterface.h`
- `frameworks/base/include/utils/Binder.h`
- `frameworks/base/include/utils/BpBinder.h`
- `frameworks/base/include/utils/IBinder.h`
- `frameworks/base/include/utils/Parcel.h`
- `frameworks/base/include/utils/IPCThreadState.h`
- `frameworks/base/include/utils/ProcessState.h`
- `frameworks/base/libs/utils/Binder.cpp`
- `frameworks/base/libs/utils/BpBinder.cpp`
- `frameworks/base/libs/utils/IInterface.cpp`
- `frameworks/base/libs/utils/ProcessSTate.cpp`
- `frameworks/base/libs/utils/IPCThreadState.cpp`

5.4. C Kernel Driver

The Binder kernel driver is the heart of the Binder framework. At this point, the reliable and secure delivery of messages must be guaranteed. The kernel driver is a small kernel module and is written in C. The driver module is build from the source files:

- `/drivers/staging/android/binder.c`
- `/drivers/staging/android/binder.h`

The Binder kernel driver supports the file operations `open`, `mmap`, `release`, `poll` and the system call `ioctl`. This operations represent the interface by that the higher layers access the Binder driver. The Binder operation `open` establishes a connection to the Binder driver and assign it with a Linux file pointer, whereas the `release` operation closes the connection. The `mmap` operation is needed to map Binder memory. The main operation is the system call `ioctl`. The higher layers submit and receive all information and messages by that operation. The `ioctl` operation takes as arguments a Binder driver command code and a data buffer. These commands are:

BINDER_WRITE_READ is the most important command, it submits a series of transmission data. The series consists of multiple data as described in Figure 4.3.

BINDER_SET_MAX_THREADS sets the number of maximal threads per process to work on requests.

BINDER_SET_CONTEXT_MGR sets the `context manager`. It can be set only one time successfully and follows the first come first serve pattern.

BINDER_THREAD_EXIT This command is sent by middleware, if a binder thread exits.

BINDER_VERSION returns the Binder version number.

The facilities implemented in the Binder driver are discussed in the next sections. The commands used in these section are target commands. Even if the name is target command, some of these commands are applied to the local binder. In the OpenBinder documentation they are referred to as *Binder driver protocol*. The available codes are discussed as follows and start with a `BC_` prefix for *Binder command*. The Binder diver talks back with codes starting with a `BR_` prefix for *Binder return*.

5.4.1. Binder Thread Support

Since the kernel driver does not implement the thread start mechanism, it must be kept up to date about how many threads are started. These commands are sent so that Binder driver can have an accurately count of the number of looping threads available. The target commands are `BC_REGISTER_LOOPER`, `BC_ENTER_LOOPER` and `BC_EXIT_LOOPER`. These commands are for bookkeeping and are destined for the local Binder.

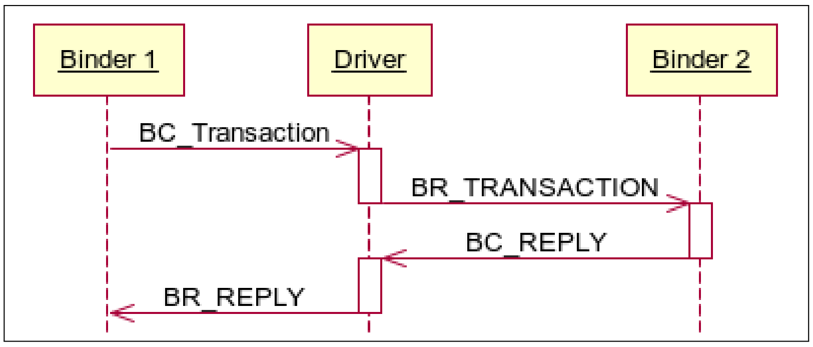


Figure 5.3.: Binder Driver Interaction

5.4.2. Binder Transactions

The commands BC_TRANSACTION and BC_REPLY cause a transit of data to another Binder interface. The BC_REPLY command is used by the middleware to answer a received BC_TRANSACTION. Figure 5.3 presents the interaction with the Binder driver when a Binder transmits a transaction and waits until receiving a reply to this transaction. The Binder driver takes care of delivering the reply to the waiting thread, that it can see the reply as direct response.

The Binder driver copies the transmission data from the user memory address space of the sending process to its kernel space and then copies the transmission data to the destination process. This is achieved by the copy_from_user and copy_to_user command of the linux kernel. The data transaction is presented in Figure 5.4.

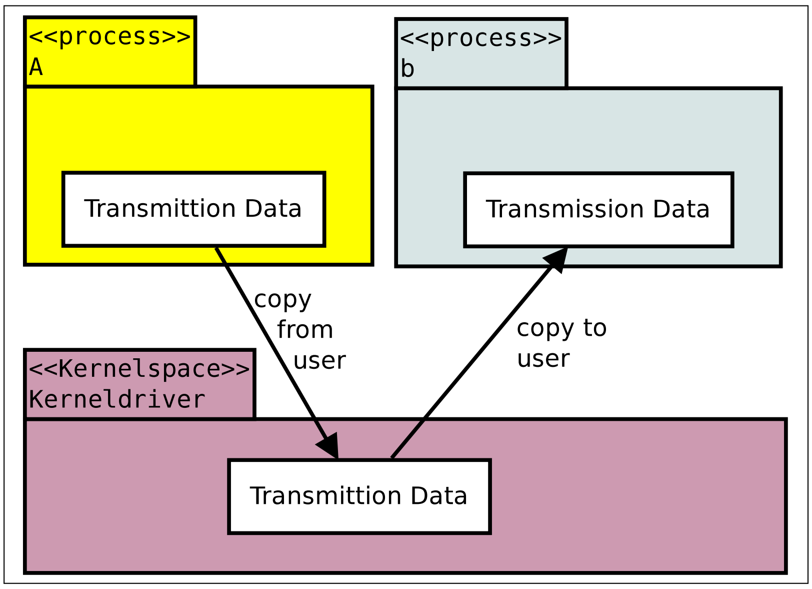


Figure 5.4.: Data Transaciton

5.4.3. Further Mechanism

The Binder kernel commands BC_INCREFS, BC_RELEASE and BC_DECREFS implement the reference counting facilities of the Binder framework. The *link to death* or *death notification* feature is also implemented in the kernel driver. The kernel driver manages and keeps all information, that are necessary to recognize and deliver the termination of a Binder node. The commands are BC_REQUEST_DEATH_NOTIFICIATION, BC_CLEAR_DEATH_NOTIFICATION, BC_DEAD_BINDER_DONE and their response codes.

6. Example IPC Message Flow

6.1. Testing Environment

We used two testing apps running on a virtual device. The first one was an mod-

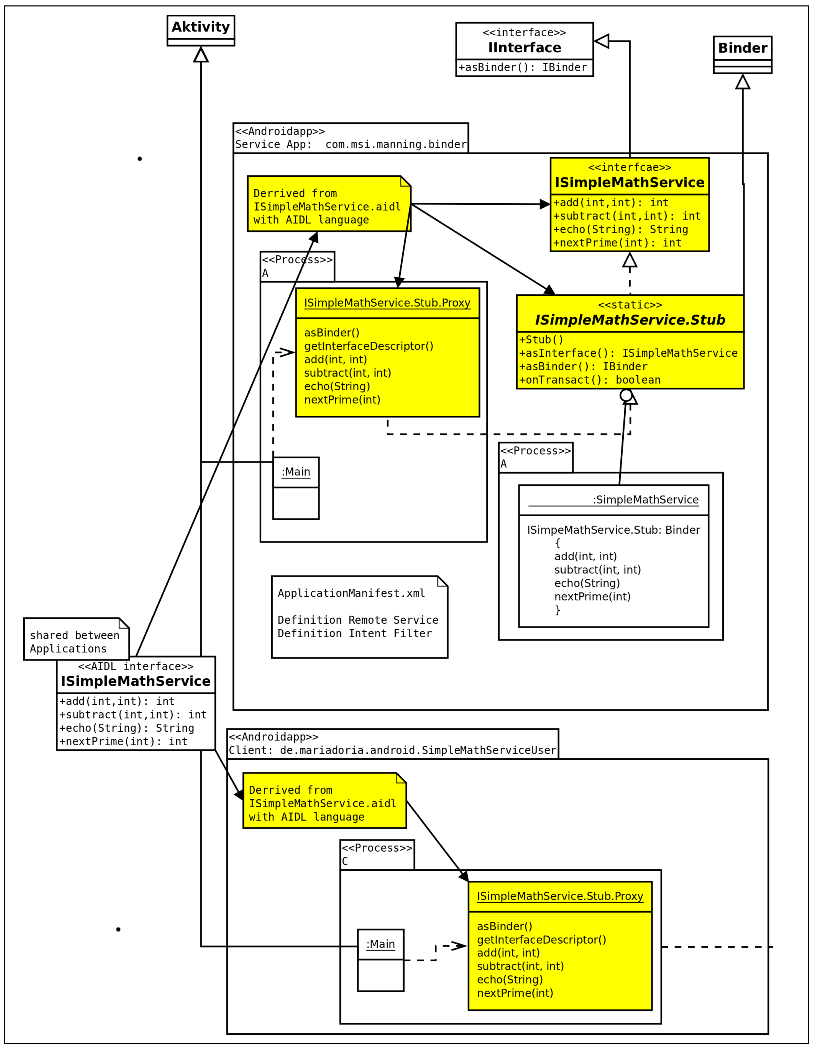


Figure 6.1.: Testing Environment

ified example[1], which was originally designed to demonstrate an in-app service communication. This is handled quite differently and copes with intercomponent communication, not interprocess communication. It is called `SimpleMathService` and offers methods for simple mathematical operations.

The second application was self-programmed and simply uses the remote service provided by the first app.

The applications of the testing environment are illustrated as an extended UML class diagram in Figure 6.1. The UML diagramm is extended with a view of the processes, which instantiated nested Java objects. Their classes and objects (shown in yellow) are compiled from the Android AIDL language. The service app has two components, an activity component called `Main` and a service component called `SimpleMathService`. The `SimpleMathService` class holds an extended anonymous inner class implementing the business logic of the service. The inner class is derived from `ISimpleMathService.Stub` which is generated by Android SDK from the ISimpleMathService.aidl definition file. This file contains the API definition of the service and must be published if other application developers wish to use this service. The stub class extends the Binder class as well as the proxy class. Accordingly, they are the endpoints of the Binder communication progress.

The application manifest declares the service as remote and therefore it is started in an own process by Android. The activity component will hold a proxy object which is a nested class of the interface `ISimpleMathService`. The activity runs in an own process, too.

The second application contains the `Main` activity as component only. This main object holds the UI with a button and a text output and a proxy object for the `SimpleMathService`, that is more related to IPC.

The applications were compiled with Android SDK for Eclipse and were executed in an Android emulator.

6.2. Message Flow and Call Stacks

Due to the limited size of this paper, only an excerpt can be presented. The binding of the service is presented abstractly, and the remote procedure call is presented in detail.

The user app is executed first. It asks the *service manager* for a Binder of the `SimpleMathService`. This Binder is implemented as an anonymous class from `ISimpleMathService.Stub` of the `SimpleMathService` application. In this example, the stub object implements the business logic of the service. The

[1] from "unlocking android": com.manning.binder

`bindService()` method of the `Main` class of the user app will create a proxy object for the requested service, which communicates with the stub object on the server side.

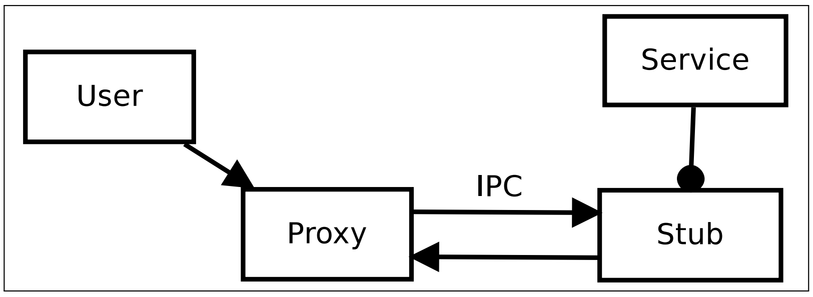


Figure 6.2.: Proxy and Stub

At this point we describe in detail what happens, when a remote procedure is called after the binding to the service has been established. The listing 6.1 shows a nested anonymous extended class, which is used as callback.

```java
ISimpleMathService mService;
private ServiceConnection mConn = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder
        service) {
      mService = ISimpleMathService.Stub.asInterface(service);

    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
      mService = null
    }

  };
```

Listing 6.1: Callback on Connection Events

This `ServiceConnection` object was an argument of the earlier called `bindService()` method. The `onServiceConnected` callback method returns a proxy object, which delivers all method calls to the remote service. The user application can now handle the object as if it were a local object and marshal methods.

```java
int i =40; int j = 2;
mService.add(j,i);
```

Listing 6.2: Remote Method Call

Listing 6.2 does a method call on the proxy object to add 40 and 2. In the background, following happens: The call is divided by the proxy object in 6.3 into basic data types, which can be written in a parcel. At first, the receiver is written to the parcel, that is a Binder. The arguments are written serialized in the data packet. A user defined int code is assigned to the transaction. This code relates to the intended method name, because the Binder framework at this point permits only to submit an integer value. To avoid misunderstandings, the remote service as the user application must use the same assignment of code and methods.

```java
public int add(int a, int b) throws android.os.RemoteException
{
  android.os.Parcel _data = android.os.Parcel.obtain();
  android.os.Parcel _reply = android.os.Parcel.obtain();
  int _result;
  try {
    _data.writeInterfaceToken(DESCRIPTOR);
    _data.writeInt(a);
    _data.writeInt(b);
    mRemote.transact(Stub.TRANSACTION_add, _data, _reply, 0);
    _reply.readException();
    _result = _reply.readInt();
  }
  finally {
    _reply.recycle();
_   data.recycle();
  }
  return _result;
}
```

Listing 6.3: Proxy Method

At this point, the interprocess communication is initiated with the transact method.

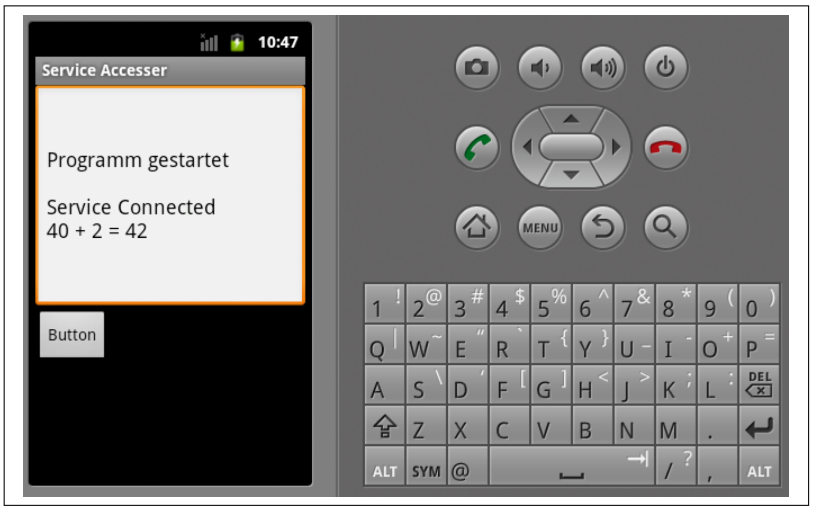
The parcel is sent to the JNI interface that sends it to the Binder C++ middleware that sends it to the Binder kernel driver. The Binder kernel driver will send the client process to sleep and map the parcel data and the code from client process to the server process. The parcel is send from Binder driver to C++ middleware and then to JNI and on Java API Wrapper Layer the method `ontransact` of the stub is called.

```java
@Override public boolean onTransact(int code, android.os.Parcel data
    , android.os.Parcel reply, int flags) throws android.os.
    RemoteException
{
switch (code)
{
case TRANSACTION_add:
{
data.enforceInterface(DESCRIPTOR);
int _arg0;
_arg0 = data.readInt();
int _arg1;
_arg1 = data.readInt();
int _result = this.add(_arg0, _arg1);
reply.writeNoException();
reply.writeInt(_result);
return true;
}
}
```

Listing 6.4: Stub Method

In Listing 6.4 the entry point for receiving a message is presented. The code is read first and due to knowledge of the method signature the accurate count of arguments are read from the parcel. Now the method corresponding to the code implementing the business logic is called with extracted arguments. The result is written to a reply parcel.

Again it is routed through the layers to the binder driver, that transfers the parcel and wakes up the sleeping client process and delivers the reply parcel to the proxy object. The relpy is unparceled and returned as the result of the proxy method. Thereafter the result is displayed at the activity window of the client app, refer Figure 6.3.



Programm gestartet

Service Connected
40 + 2 = 42

Figure 6.3.: Testing Environment

7. Discussion

The Binder framework supports basic security features. It ensures that no other application can read or manipulate data by transmitting them over a private channel, namely the Binder kernel module. It acts as mediator and must be trusted by the communicating parties. For identification, the Binder framework provides the UID and PID of the calling Binder. With the UID, an application can check the package signature and identify the app. [7] This is important, because multiple services can be assigned with the same name. The operating system will decide, which service is called, depending on the set priority of the service. However, it is possible for a malicious service to overlap the good service and retrieve information, that is sent by the App believing it is communicating with a trusted service. The application must ensure in security critical situations, e.g. the login to a service, the identity of the service. This is possible and this work could not find a flaw in that system, since the UID and PID are derived from Linux methods, that can be regarded as secure and can not be manipulated by unintended calls or arguments.

The use of Binder as a security token should be audited, because the binder reference number is not chosen randomly. It is incremented from zero in the Binder driver. It could be possible to increase the possible numbers and guess with good probability the right Binder token. But this must be confirmed in a future work.

A. Bibliography

[1] Openhandset Alliance. Android overview, 08 2011. URL http://www.openhandsetalliance.com/android_overview.html.

[2] Bornstein. Dalvik vm internals, 2008 google i/o session, 01 2008. URL http://sites.google.com/site/io/dalvik-vm-internals.

[3] Brady. Anatomy & physiology of an android, 2008 google i/o, 2008. URL http://sites.google.com/site/io/anatomy--physiology-of-an-android.

[4] Winandy Davi, Sadeghi. Privilege escalation attacks on android, 11 2010. URL http://www.ei.rub.de/media/trust/veroeffentlichungen/2010/11/13/DDSW2010_Privilege_Escalation_Attacks_on_Android.pdf.

[5] David Ehringer. Dalvik virtual machine, 03 2011. URL http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf.

[6] Enck. Understanding android security. *IEEE S*, JanuaryFebruary:50pp, 2009.

[7] freyo. Android get signature by uid, 07 2010. URL http://www.xinotes.org/notes/note/1204/.

[8] Gartner. Gartner says android to become no. 2 worldwide mobile operating system in 2010 and challenge symbian for no. 1 position by 2014, 10 2010. URL http://www.gartner.com/it/page.jsp?id=1434613.

[9] Google. Android java sources, .

[10] Google. Android kernel sources, .

[11] Google. Android interface definition language (aidl), 08 2011. URL http://developer.android.com/guide/developing/tools/aidl.html.

[12] Google. Android documentation - fundamentals, 08 2011. URL http://developer.android.com/guide/topics/fundamentals.html.

[13] Google. The android mainifest xml file, 08 2011. URL `http://developer.android.com/guide/topics/manifest/manifest-intro.html`.

[14] Google. Binder java documentation, 08 2011. URL `http://developer.android.com/reference/android/os/Binder.html`.

[15] Google. Android documentation - intent, 08 2011. URL `http://developer.android.com/reference/android/content/Intent.html`.

[16] Google. Android security, 08 2011. URL `http://developer.android.com/guide/topics/security/security.html`.

[17] Goolge. Android documentation - what is android, 08 2011. URL `http://developer.android.com/guide/basics/what-is-android.html`.

[18] Security Engineering Research Group. Android security, a survey. so far so good., 07 2010. URL `http://imsciences.edu.pk/serg/2010/07/android-security-a-survey-so-far-so-good/`.

[19] Hackborn. Re: [patch 1/6] staging: android: binder: Remove some funny && usage, 06 2009. URL `https://lkml.org/lkml/2009/6/25/3`.

[20] Palmsource Inc. Open binder documentation, 12 2005. URL `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html`.

[21] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel, 2011.

[22] Oracle. Java native interface, 08 2011. URL `http://download.oracle.com/javase/6/docs/technotes/guides/jni/index.html`.

[23] David A Rusling. *The Linux Kernel*. 1999.

[24] Chin Felt Greenwood Wagner. Analyzing inter-application communication in android, 06 2001. URL `www.cs.berkeley.edu/~afelt/intentsecurity-mobisys.pdf`.

[25] Wiki. Android memory usage, 08 2011. URL `http://elinux.org/Android_Memory_Usage`.