Universitá di Pisa

Laurea Magistrale in Computer Engineering

Dipartimento di Ingegneria dell'Informazione

Jarvis: Bridging the Semantic Gap between Android APIs and System Calls

Supervisors:

Chiar.mo Prof. Gianluca Dini
Chiar.ma Prof.ssa Cinzia Bernardeschi
Chiar.mo Prof. Giovanni Vigna

Candidate:

Tommaso Latini

University Year 2013-2014

Abstract

Android is an open-source operating system, based on the Linux kernel. Developed by Google, it is currently the most widespread mobile operating system, used by more than one billion users.

Different solutions have been proposed to analyze the behavior of Android's applications. Most of this approaches analyze applications at a high-level (e.g., they focus on the executed Java code) and they may miss operations performed by applications' components natively written. For this reason, other approaches analyze applications' behavior at the system-call level. However, given the complexity of the Android runtime library, using this low-level information to infer high-level behaviors is problematic.

To bridge the semantic gap between high-level Android APIs and low-level system calls, we have developed Jarvis. Jarvis operates in two phases: an online data collection phase, and an offline data analysis phase.

Specifically, during the online data collection phase, Jarvis records all the system calls executed by an application. In this phase, particular care is given to system calls used to communicate with Binder. Binder is an Android kernel module that manages most of the Inter-Process Communication among installed applications and system components. Since, in Android, Binder is extensively used by every application to communicate with the operating system, analyzing data exchanged using it is crucial to understand the behavior of an application. For this reason, Jarvis automatically parses data sent and received by an application using Binder. In particular, Jarvis is able to understand the origin and the end-point of every Binder-mediated transaction and reconstruct the high-level representation of the exchanged data.

During the data collection phase ad-hoc filters can be specified to precisely define when Jarvis should record the execution of a specific system call. In this way, the performance impact of Jarvis is substantially reduced.

Finally, in the data analysis phase, Jarvis maps how different APIs produce specific lists of system calls.

We performed different experiments to evaluate Jarvis. Results show the effectiveness of our approach, but also reveal some issues in the current implementation. For this reason, we conclude this work by suggesting different ways to address these issues and improve the capabilities of Jarvis.

Contents

Chapter 1

Introduction

Android managed to grab around 84.6% smartphone share in the second quarter of 2014 (corresponding to 295.2 million units), up from the 80.2% of the previous year (around 249.6 million units). Rounding out the top four smartphone operating systems were iOS at 11.9%, Windows Phone at 3.6%, and BlackBerry at 1.7%.

1.1 Background and Context

The Figure 1.1 shows an indisputable leadership that motivated malware authors to work on Android. Indeed the US Department of Homeland Security and FBI [6] shows that almost the 80% of the known malware threats to mobile operating system in 2012 concerns Android.

Moreover, according to recent research a percentage between 0.02 and 0.2 of applications in official and unofficial marketplaces are malicious [22]. Corporations and Researchers needs tools as powerful as possible to analyse the behavior of the operating system both in presence and in absence of malware to test the safeness and the robustness.

1.1.1 Malware Analysis

We can roughly define a **Malware** - short for **Mal**icious soft**ware** - any program (executable or script) designed and built to damage a computer or a network, to gather sensitive information or to gain access to private computer systems. The malware category comprehends computer viruses, worms, trojan horses, spyware and other malicious program.

"*Malware Analysis* is the art of dissecting malware to understand how it works, how to identify it, and how to defeat or eliminate it" [15]. It can be done using two complementary techniques:

- **Static Analysis**, which studies a program without executing it.

Global Smartphone Operating System Shipments (Millions of Units)	Q2 '13	Q2 '14
Android	186.8	249.6
Apple iOS	31.2	35.2
Microsoft	8.9	8.0
BlackBerry	5.7	1.9
Others	0.5	0.5
Total	233.0	295.2

Global Smartphone Operating System Marketshare %	Q2 '13	Q2 '14
Android	80.2%	84.6%
Apple iOS	13.4%	11.9%
Microsoft	3.8%	2.7%
BlackBerry	2.4%	0.6%
Others	0.2%	0.2%
Total	100.0%	100.0%

Total Growth Year-over-Year %	48.9%	26.7%

Source: Strategy Analytics

Figure 1.1: Smartphone Sales compared with Last Two Years

Static analysis exploits disassemblers, decompilers, source code analyzers and other similar tools. It has the advantage to reveal the behavior of the program also in unusual condition, but the results could be imprecise, especially for big programs. Moreover, *Obfuscation* [1] makes it really arduous.

- **Dynamic Analysis**, which examines the behavior of a program during its execution.

  Dynamic analysis is often faster and more accurate and allows a realistic simulation of the programs. It's necessary for a precise diagnosis of runtime information. The main problem is that you only see operations that the program did during execution or rather you don't know "all the possible things that the program could do".

  It is useful because of two features mainly:

  - **Precision of Information**: tool used for dynamic analysis can in general collect precisely data needed to address the problem;
  - **Dependence on Input**: changes in program input are directly observable in terms of output and internal behavior [1].

Doing dynamic analysis in a safe way (it is a matter of executing potential malicious code) requires the program to be confined in a safe environment

[1] Obfuscation is a process that transforms the source code in an "intermediate" code to make it more difficult to be decompiled and disassembled using classical reverse engineering techniques.

generally provided by a virtual machine [2]. Other techniques isolate the program at process-level so the interface of kernel is shared between the "target" program and the other ones. An example of jail technique is the UNIX `chroot()` utility.

1.1.2 System Call Monitoring

System Calls Monitoring is a widely used technique for program analysis and consists in looking at information that crosses the boundary between user and kernel space. This edge can be traverse only by system calls, characterized firstly for a names, some arguments, and a possible result values. In between system calls what happens within a process is completely ignored. The entire process is treated as a black box. This approach makes sense in operating environments where even simple and common operations requires assistance by the kernel [3].

Android falls in the previous case because of common requirements of mobile operating system. For example, every data exchange between two applications and every request of a system service exploits an Inter-Process Communication mechanism whose name is *Binder*, which in turn implies one or more `ioctl()` call on the corresponding driver. Android kernel is based on Linux: most of the functions are the same excepts for some mobile system specific features as *Power Management* or *Low Memory Killer*.

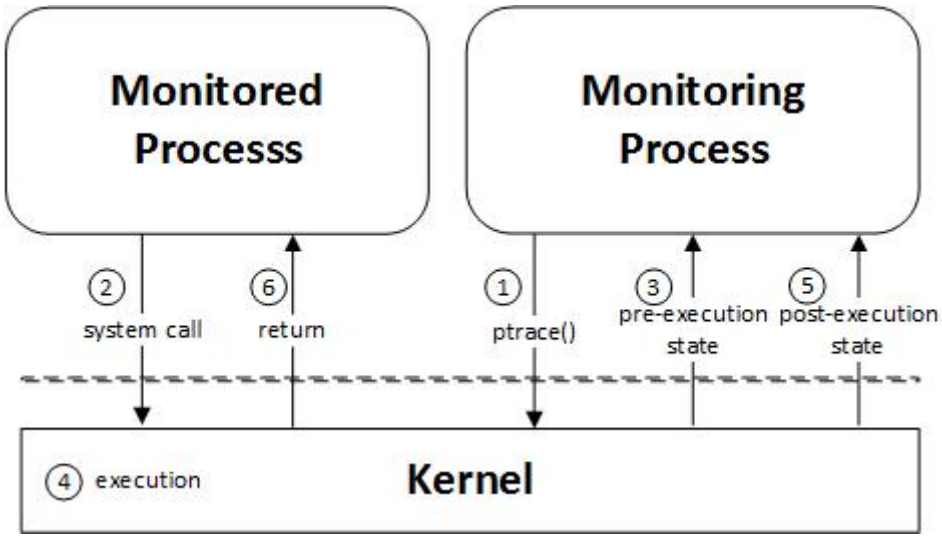

Figure 1.2: Control flow with a System Call Monitor

Modern UNIX-like operating systems provide tools for monitoring sys-

[2] A *virtual machine* creates a complete system platform that supports the execution of an operating system.

tem calls in real time like `strace` and now there exists also a version for Android. The underlying mechanism is based on the `/proc` file system or the `ptrace()` system call. Figure 1.2 briefly summarizes the general behavior of this kind of tool:

- A process invokes a system call;
- The monitor inspects the state of the system and then returns the control to the kernel;
- The kernel executes the system call;
- The monitor inspects newly the state of the system;
- The kernel passes control back to the monitored process.

The monitoring process generally resides in the user space and uses `ptrace` to control the flow the monitored one.

1.2 Objectives

This dissertation exhibits the results of the work done at Computer Security Lab (SECLAB) - University of California, Santa Barbara (UCSB), between August and October 2013. The goal of the project was to build a tool allowing to bridge the semantic gap between high-level Android *APIs* [3] and low-level System Calls. It required a strong analysis of Android specific mechanisms like the Binder - the component used to exchange data between application - and the design, implementation and use of a tool that compounded a System Call Monitor with Filtering and Data Interpretation mechanisms.

To summarize, the project encompassed the documentation and formalization of low-level features of Android Binder and a tool for logging of Binder request/response at low-level and rebuilding of structured data then enlarged to track general system calls in order to create a tool that allowed an automatic mapping between Android APIs and System Calls.

This kind of mapping can be used to associate a "signature" in terms of system calls to an API and to use it to analyse high-level behavior of application.

This is the first work that tries to understand if it is possible to automatically reconstruct the high-level behavior of an Android application.

[3] API, an abbreviation of Application Programming Interface, refers either to a set of routines, protocols, and tools for building software applications or to a single available pre-defined function. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together.

1.2.1 Related Work

Tracking system call invocations is at the basis of virtually all the dynamic malware behavioral analysis systems [2]. The reconstruction of high level behavior from system calls is widely used in computer security for intrusion detection [4], [13], [19]. Nevertheless, standard tools implement virtual environment based on x86 CPU and cannot be used on Android because it runs on ARM processors. Moreover, some peculiar characteristics (an improved Linux Kernel, a particular version of Java Virtual Machine) makes this operating system different from those of the UNIX-like family. The nature of Android applications makes it hard if not impossible to rely on existing VM-based dynamic malware analysis systems as is, because it can be implemented either in Java, at the top of DVM, or in C++, using JNI.

Some tools has been implemented to fill this gap. For instance, *CopperDroid* - an approach built on top of QEMU - performs dynamic behavioral analysis of Android malware [7]. It presents a unified analysis to characterize low-level OS-specific and high-level Android-specific behaviors.

However, none of the previous tools provides a well-defined mapping between Android APIs and Sytem Calls. This is the key point of our thesis: verifying the possibility to bridge the semantic gap between high-level and low-level world in Android.

1.2.2 Approach

We decided not to use or modify `strace` tool because our research was initially focused on Binder, so we started creating an `ioctl()` interceptor that was well-integrate with Java to perform unmarshalling on logged data, then we enlarged the tool for logging other system calls. Moreover, some peculiar features of Android applications and framework forced us to build a tool that resides into the kernel.

We firstly focused on system calls related to process control (`fork()`, `execve()`, `clone()`), networking (`socket()`, `bind()`, `connect()`) and file-system management (`open()` `access()`, `mkdir()`). We implemented also a filter to clean the log by the noise (i.e. system calls not directly related to a specific behavior).

1.3 Work Organization

This document is organized into three parts and ten Chapters (including this introduction):

Part I :

We briefly introduce the main characteristics of Android operating systems, such as the architecture, the basic structure of the applica-

tion and the security mechanisms. Then we focus our attention on the Binder, describing how this IPC mechanism works from the Java utilities (the highest level) to the kernel module (the lowest level), and showing all the facilities provided by the framework. Eventually, we explain the low-level communication protocol used by Binder framework to interact with the driver and we deepen some implementation details concerning key aspects of Binder.

Part II :

We describe the tool Jarvis, the main components, the working principles and the design choices, then we deepen in the implementation of Jarvis, focusing on the modules and on some peculiar programming aspects, at last we explain how to exploit the tool and how to improve it by means of ready-to-use template.

Part III :

We show an initial attempt of mapping APIs and system calls in order to prove the right behavior of the tool, drawing some conclusions specifying where and how our tool can be useful in future researches and which possible improvements can be easily implemented.

Part I

Android Operating System and Binder IPC

Chapter 2

Android Overview

In this Chapter we make a brief presentation of Android. We show the architecture of the whole system, then the basic structure of applications with their main components and the way they communicate each other. Eventually we describe the most important security mechanisms.

2.1 Android Architecture

Android is a mobile operating system based on the Linux kernel, with middleware, libraries and APIs written in C++ and application software running on a framework that includes a non standard register-based Java virtual machine specific for ARM hardware - called *DVM* [1] - with limited implementation of Java libraries. It utilizes just-in-time compilation to run *dex* [2] code, which is usually obtained from Java bytecode. In Figure 2.1, it's easy to distinguish three main parts:

- The application level [Blue];
- The middleware [Green];
- The kernel [Red].

The application level invokes Java APIs, which rely on the C++ middleware by means of binding libraries. Dalvik Virtual Machine executes its own bytecode so the application cannot directly invokes native code unless using the *JNI* [3], which is a wrapper for the underlain C++ functions and allows applications to call the methods of the various frameworks without using android runtime environment. This feature of Android provides lots

[1] Dalvik Virtual Machine.
[2] **D**alvik **ex**ecutable.
[3] Java Native Interface.

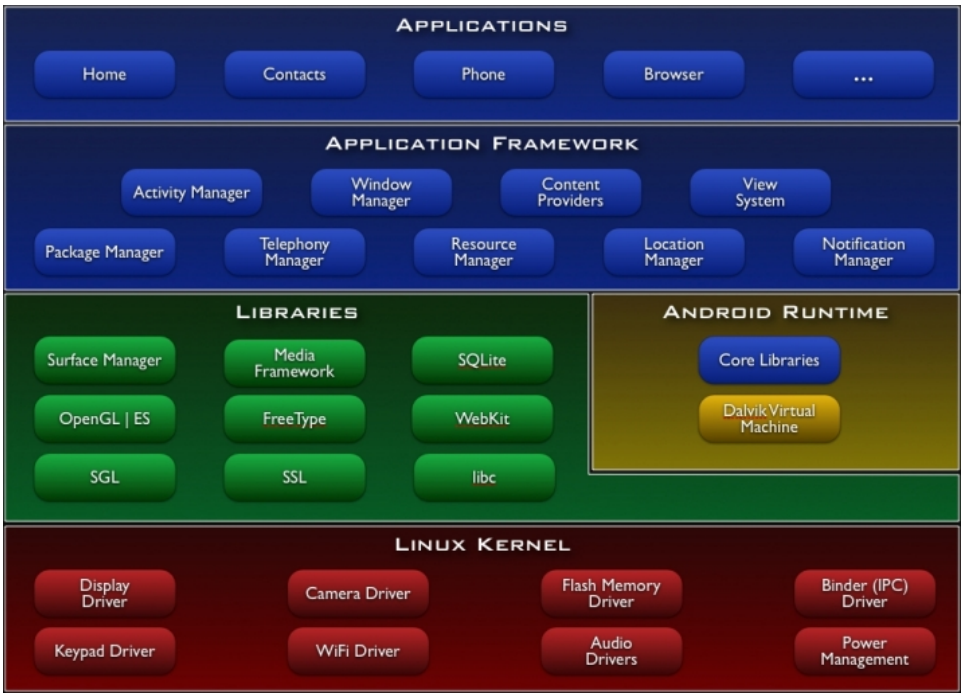

Figure 2.1: Android Architecture

of different design choices to the developer, which can directly interact with low-level APIs.

The middleware uses directly the Linux System Call Interface (SCI) to interact with the kernel. The peculiar features of Android kernel are implemented as device drivers and are reachable by `ioctl()` system call, so the structure and the general organization of the kernel is preserved and does not differ a lot with respect to normal distributions.

2.2 Structure of Application

The structure of a genericAndroid application is shown in Figure 2.2. We can identify four main components [8]:

- **Activities**

  An activity represents a single screen with a user interface. Each application is independent of the others, hence another application can start any one of these activities (if it has the permissions). For example, an email application might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails.

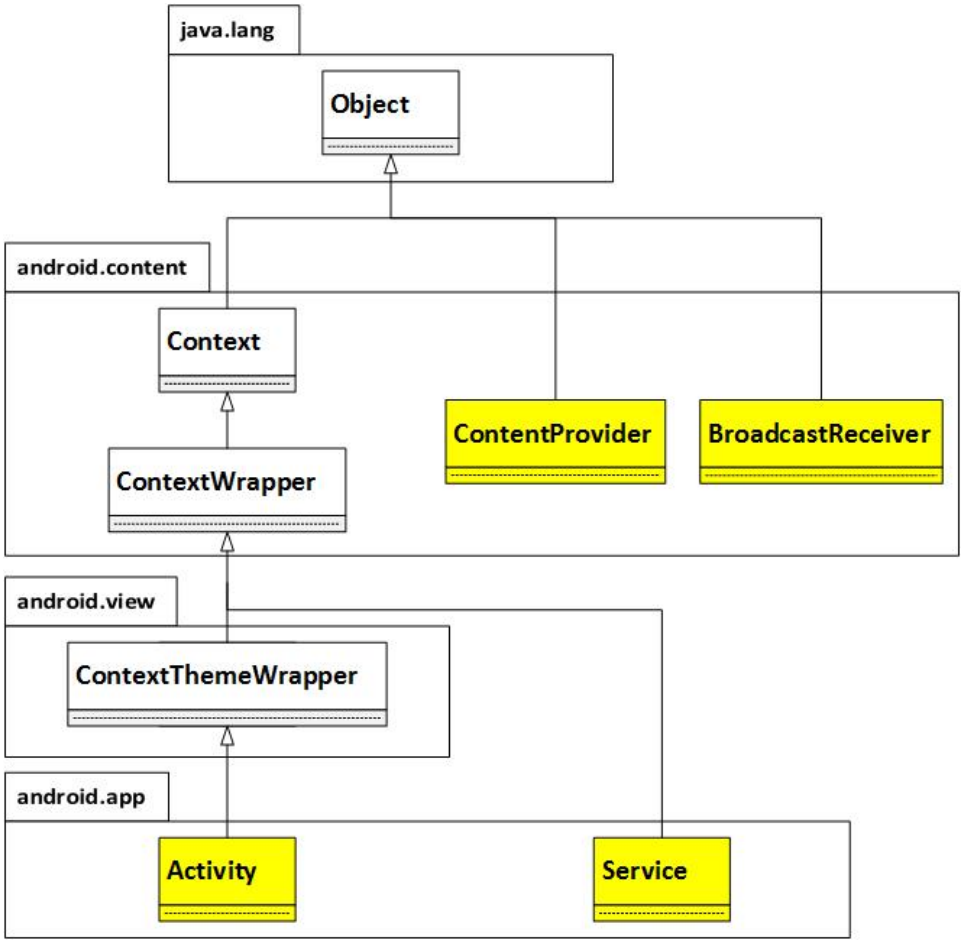

Figure 2.2: Main Components of an Android Application

- **Services**

  A service is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different application.

- **Content Providers**

  A content provider manages a shared set of application data. You can store the data in the file system - an SQLite database - on the web, or any other persistent storage location your application can access. Through the content provider, other applications can query or even modify the data (if the content provider allows it). For example, the Android system provides a content provider that manages the users'

contact information.

- **Broadcast Receivers**

  A broadcast receiver is a component that responds to system-wide broadcast announcements. For example, a broadcast announcing that the screen has turned off, the battery is low, or a picture is captured.

By default, all components of the same application run in a single process and thread (called the "main" thread). If an application component starts and there already exists a process for that application, then the component is started within that process and uses the same thread of execution. However, different components of the same application can run in separate processes, and there can be additional threads for any process.

2.2.1 Intent

If two components belonging to different processes or applications have to exchange data, then they have to use Intents.

An **Intent** is an high-level abstraction - underlying on Binder Framework - that hides all low-level inter-process communication mechanisms providing a simple interface for process communication. It is a data structure whose main fields are [8]:

- A **Component Name**, which declares the component to start. It is the critical piece of information that makes an Intent either *explicit*, meaning that it should be delivered only to the component defined by the name, or *implicit*, that is the system decides which component should receive the intent based on the other intent information (such as the action, data, and category—described below).
- An **URI**, which refers the data to be acted on and their *MIME* type of that data;
- An **Action**, which explains the operation to be done;
- A **Category**, which holds additional information about the kind of component that should handle the intent. Any number of Category descriptions can be placed in an intent, but most intents do not require a category;
- Some **Extras** (key-value pairs), which carry additional information required to do the requested action;
- A bunch of **Flags**, which act as a meta-data for the intent.

This Intent is submitted by the interprocess communication system. We can list four kinds of intents:

1. Starting an activity to get a result;
2. Communicating with a service (*call-back mechanism*);
3. Querying a content provider;
4. Receiving an intent broadcast (*request of contents*).

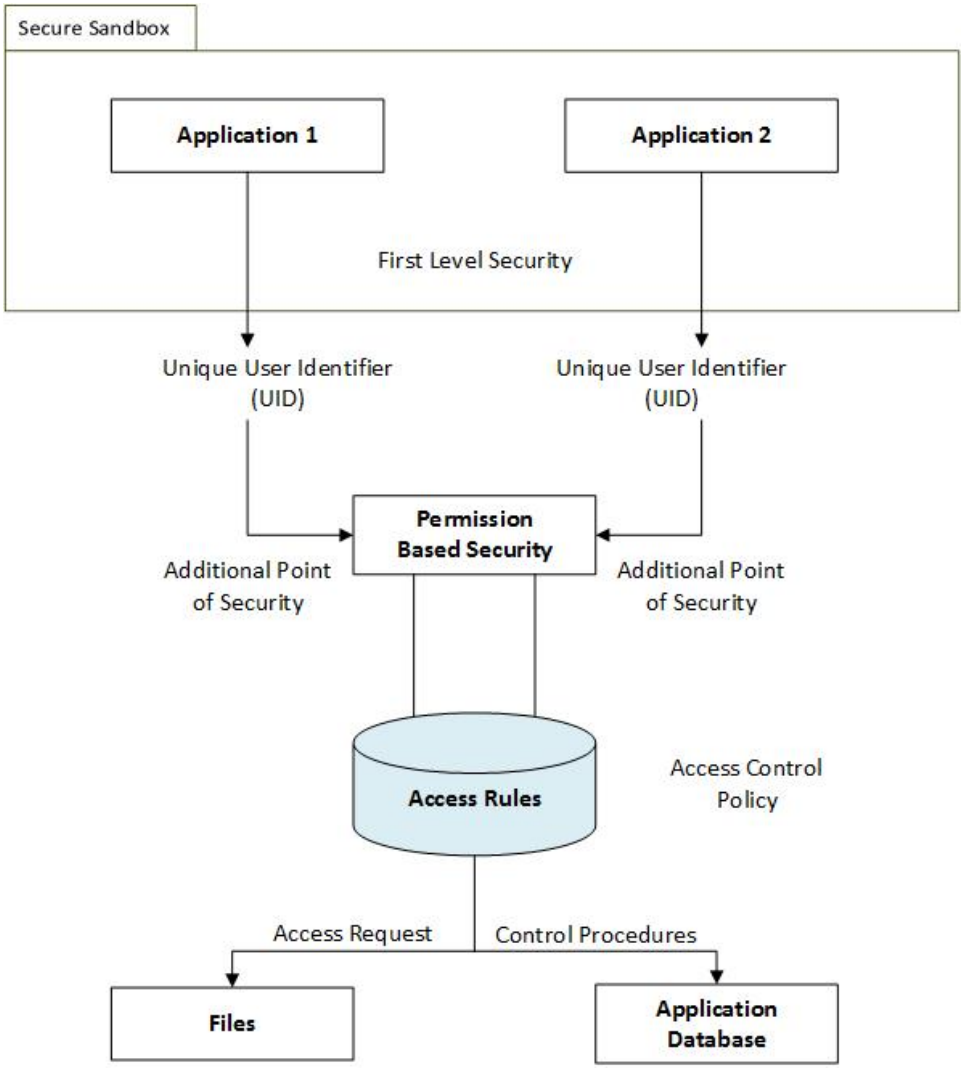

Figure 2.3: Android Security Overview

2.3 Security principles

Android implements lots of traditional security controls in order to:

- Protect user data;
- Protect resources and network;
- Provide application isolation.

Android's Applications runs on its own instances of the Dalvik Virtual Machine. Each instance represents a process completely isolated from the other application and memory. Each application of Android has a unique UID [4] and particular permissions to access the database and file on the phone. The UID is assigned also to any data stored by the application itself. Normally, other packages cannot access to these data.

This sets up a kernel-level *Application Sandbox*. The Kernel enforces security between applications and the system at the process level through standard Linux facilities, such as user and group IDs that are assigned to applications. By default, applications cannot interact with each other and applications have limited access to the operating system [11].

Android security features at the OS [5] level ensure that even native code is constrained by the sandbox. This mechanism is not heavy in overload because the Dalvik virtual machine is launched once at boot, and each instance is cloned by the first. No memory pages are copied till the application writes data on the heap. These functionalities are provided by a service called *Zygote*. Hence, Android platform provides the security of the Linux Kernel, as well as IPC[6] facility to enable secure communication among applications.

Moreover, each application must be signed by the author. The signature does not need to be of a certification authority. It's a simple private key that allow to the system to distinguish application authors to grant/deny permissions at signature-level and to answer to `shareUserId` requests.

2.3.1 Permissions mechanism

By default, an application has not any particular permission. If it wants to access to any service or data on the device, then it must require a special permission in the `AndroidManifest.xml` file.

All the requests are checking by the package installer - based on the application's signature - and by the interaction with the user. In general a permission failure launches a `SecurityException`. Once granted at installation time, a permission can never be removed except by uninstalling the application.

For example, if an application wants to receive an sms, it must write the following line on their manifest:

[4] User Identifier.
[5] Operating System.
[6] Inter-Process Communication.

```
<manifest xmlns:android="http://schemas.android.com/apk/res
    /android"
     package="com.android.app.myapp" >
     <uses-permission android:name="android.permission.
         RECEIVE_SMS" />
     ...
</manifest>
```

The user explicitly accepts during installing phase whether to give or not that particular permission to the application.

Chapter 3

Binder Framework

In this Chapter we focus our attention on **Binder Framework**: the standard IPC mechanism in Android.

This operating system uses an exchange messages communication model and the Binder Framework takes care to create the abstraction of a channel among processes. It is important to understand how this Android-specific framework works because it's responsible for any data exchange among applications. The main usage is to call remote methods exposed by system Services.

Binder is an adapted version of *Open Binder*: a software developed by *PalmSource Inc.* as a system-level component whose purpose was to provide a powerful IPC mechanism able to work in the most important operating system like Linux, Windows, BeOS.

Although the Android Binder presents some peculiar features, a good initial reading to have an overview of the whole framework is the Open-Binder documentation [14], which describes the set of facilities provided by the framework.

3.1 Binder Objects

Binder is used both for data exchange and for methods invocation as a classical RPC[1] mechanism.

A **Binder Object** is the fundamental unit of the Framework. It is a generic implementation of a **Binder Interface** (`IBinder`), which contains the list of the functions provided by a service. According to standard terminology, in the following we refers to a Binder Object simply as "binder".

An application can invoke a service only if it owns the respective binder, which is uniquely identify by means of a **Binder Token**. From a process point of view, a binder is either **local**, if the process itself created the object, or **remote**, otherwise.

[1] Remote Procedure Call

3.2 Service Manager

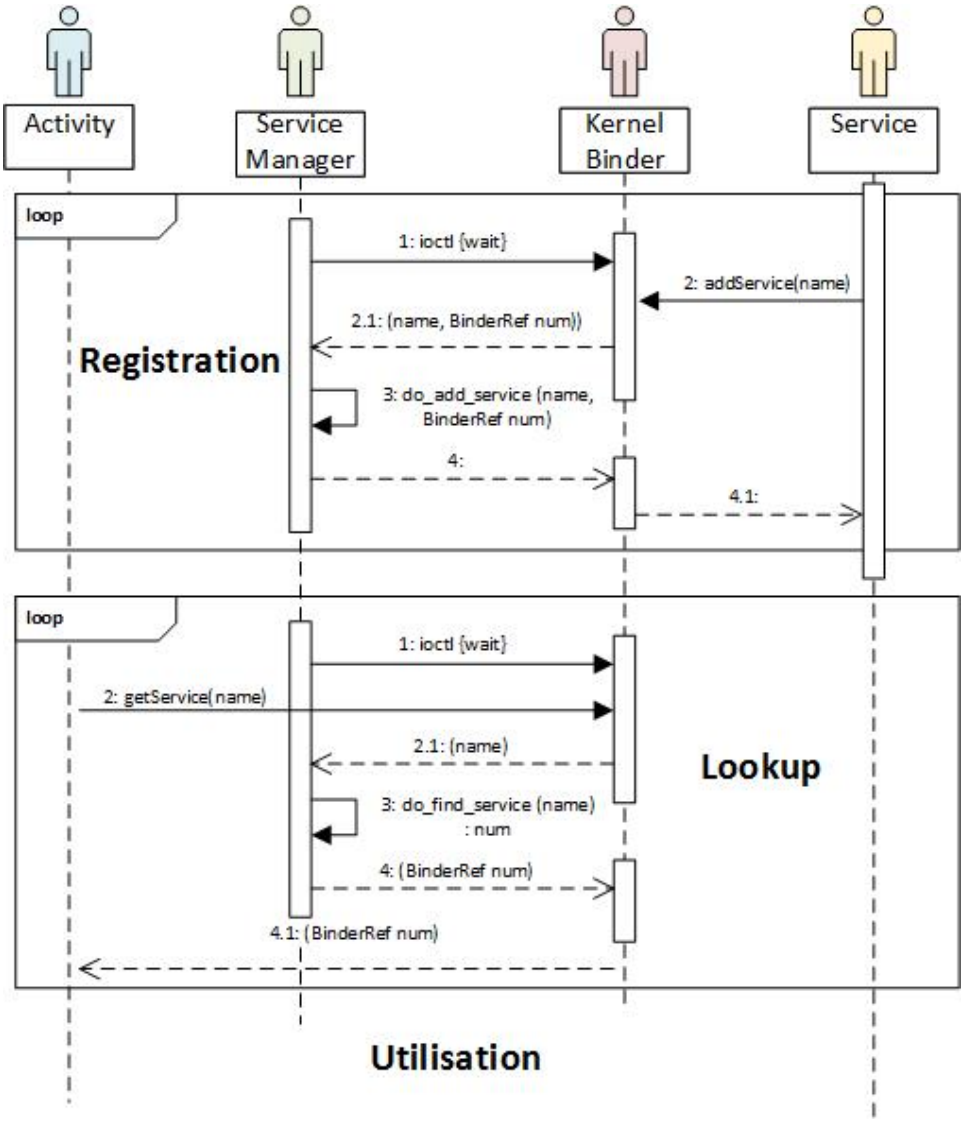

Figure 3.1: Sequence Diagram of Registration and Lookup Processes

For security issues, a process knows only the address of its local binders, so it needs some mechanism to search and recover remote binder addresses.

In *OpenBinder* terminology, **Context Manager** is the name of a special binder that have 0 as token- It is the only one to know a priori the addresses of all the binders in the system. In Android, the Context Manager takes the name of **Service Manager**. Its implementation is not part of Binder Framework [18], but it equally plays a key role in the Android IPC mechanism

because every process has to demand to Service Manager the binder of the service it wants to invoke.

Every application that provides services in Android must register itself (i.e. publish its name and its Binder Token) to the Service Manager. It can exploit the method `addService()` defined both as JAVA API and both C++ utility. The client must only know the name of a service to require the Binder address to the Service Manager (`getService()`).

3.3 Communication Model

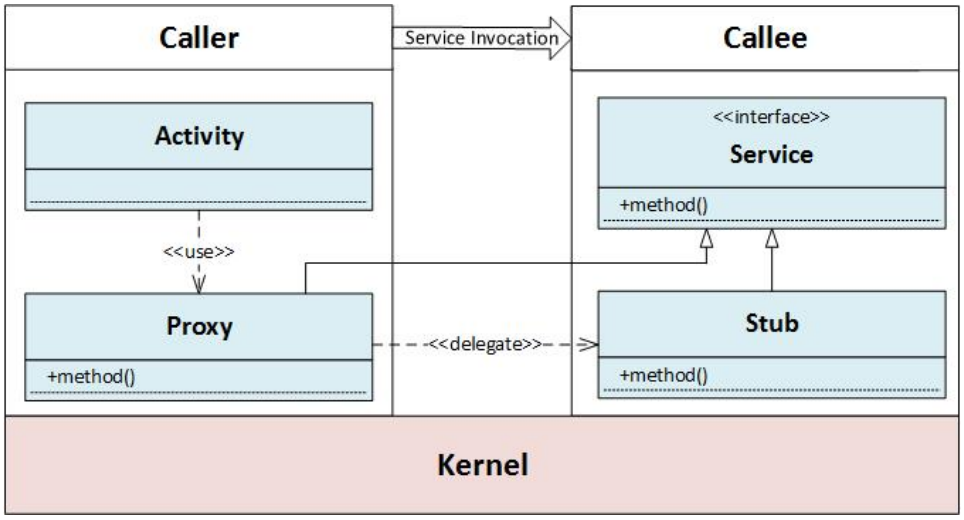

Figure 3.2: Binder Proxy-Stub model

Binder Framework uses a *proxy-stub model* (Figure 3.2) in which a proxy class creates the abstraction of a local binder, delegating to the stub the real execution of the method. In this way, an application does not need to know whether a service, a method or a file is local or remote.

3.3.1 AIDL

AIDL is the acronym of Android Interface Definition Language. Its purpose is to make easier the implementation of the service, because it automates the creation of Proxy and Stub classes. Its syntax is very close to a Java Interface - with importing library and methods signature statements - but it handles only basic types.

In a certain way, the AIDL can be equalized to Java RMI[2] in the sense

[2]The Remote Method Invocation is a mechanism that performs the object-oriented equivalent of Remote Procedure Calls (RPC), with support for direct transfer of serialized classes

that they equally automate the creation of proxy and stub classes starting from a common interface. An example of AIDL is the following [8]:

```
/** IRemoteService.aidl */
package com.example.android;

/** Declare any non-default types here with import
   statements */
/** Example service interface */

interface IRemoteService {

    /** Request the process ID of this service. */

    int getPid();

    /**
     * Demonstrates some basic types that you can use
     * as parameters and return values in AIDL.
     */

    void basicTypes(int anInt, long aLong, boolean aBoolean
       , float aFloat, double aDouble, String aString);

}
```

In the building phase, Android SDK tools generate a `.java` interface file from AIDL. The generated interface includes a subclass named `Stub` that is an abstract implementation of its parent interface and defines all the methods declared in the `.aidl` file [8]. Service developer has to take care of the implementation.

3.4 Binder Transaction and Parcel

Binder Framework provides the RMI services using **Binder Transaction**. Communication is synchronous, hence a transaction implies two messages: request and reply. Binder supports both one-way and two-way calls. On the server side, a thread pool exists for working on requests. The most important elements of a transaction are:

- the target binder;
- the method required by the client;

and distributed garbage collection. The Java original implementation depends on JVM class representation mechanisms and it thus only supports making calls from one JVM to another.

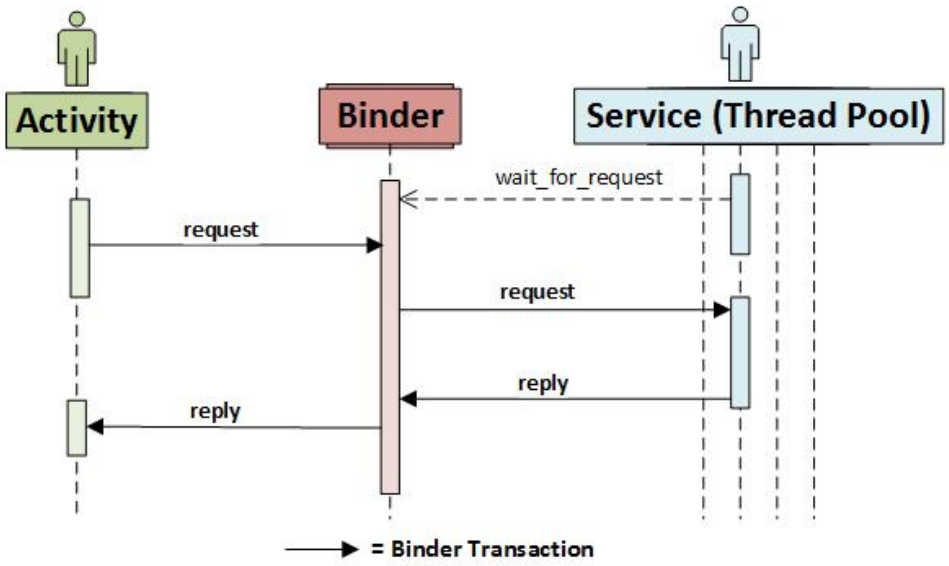

Figure 3.3: Data Transaction Schema

- the pid and the euid of sending process;
- the pointer and the size in bytes of the data buffer;

The **Parcel** is the data struture that manages transaction data. Any object that can be transmitted remotely must provide methods that serialize it on sender side and restore it on receiver side. All information must be reduced to simple data types like `integer` and `String` that can be easily written in a serial way to a buffer.

The procedure of building a Parcel is called **marshaling** or **flattening**. In reverse, the procedure of rebuilding objects from a Parcel is called **unmarshaling** or **unflattening** [18].

3.5 Other Features

Binder Framework provides other facilities that are not strictly related with our project, so we mention them briefly.

3.5.1 Death Notification

The "link to death" facility is a mechanism that allows a process to get a notification when another process that owns a binder object dies. The classic example are the window manager links to the death of a window's call-back interface. Generally services send a binder object token just to be

able to find out when client process dies. The driver notifies a process about the death of any objects it is watching.

3.5.2 Reference Counting

Binder implements a complex reference counting mechanism. Indeed, it uses two different kind of reference for a binder object:

- **strong reference**, which is a normal reference that protects the objectsfrom garbage collector;
- **weak reference**, which permits to keep trace of an object without prevent its deallocation. Java does not implement weak references hence this way to manage object lifetime is used only in the C++ middleware;

3.6 Architecture Overview

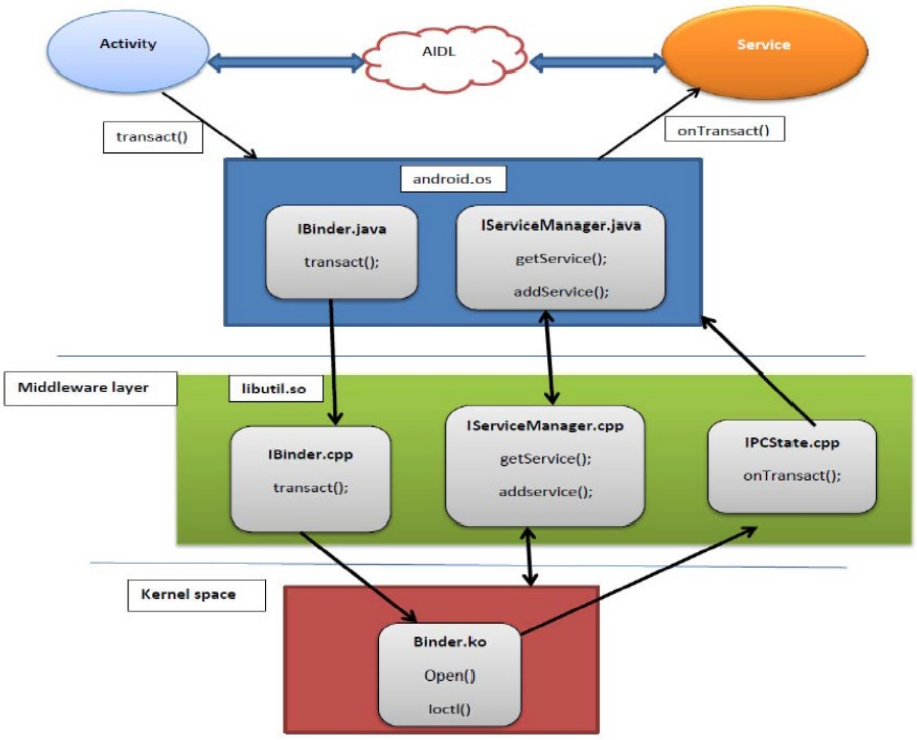

Figure 3.4: Overview of Binder Framework Architecture

Binder is organized along all the levels of Android:

- Java;
- Middleware;
- Kernel.

It uses a client-server model, in which a process (the client) initiates a communication and - in case - waits for the response, and another one (the server) that receives the request and provide to fill it. Generally, the server is a system or custom *Service* and the client is a user *Activity*.

As Figure 3.4 shows, all data exchanges between an Activity and a Service must cross the driver.

3.7 Java APIs

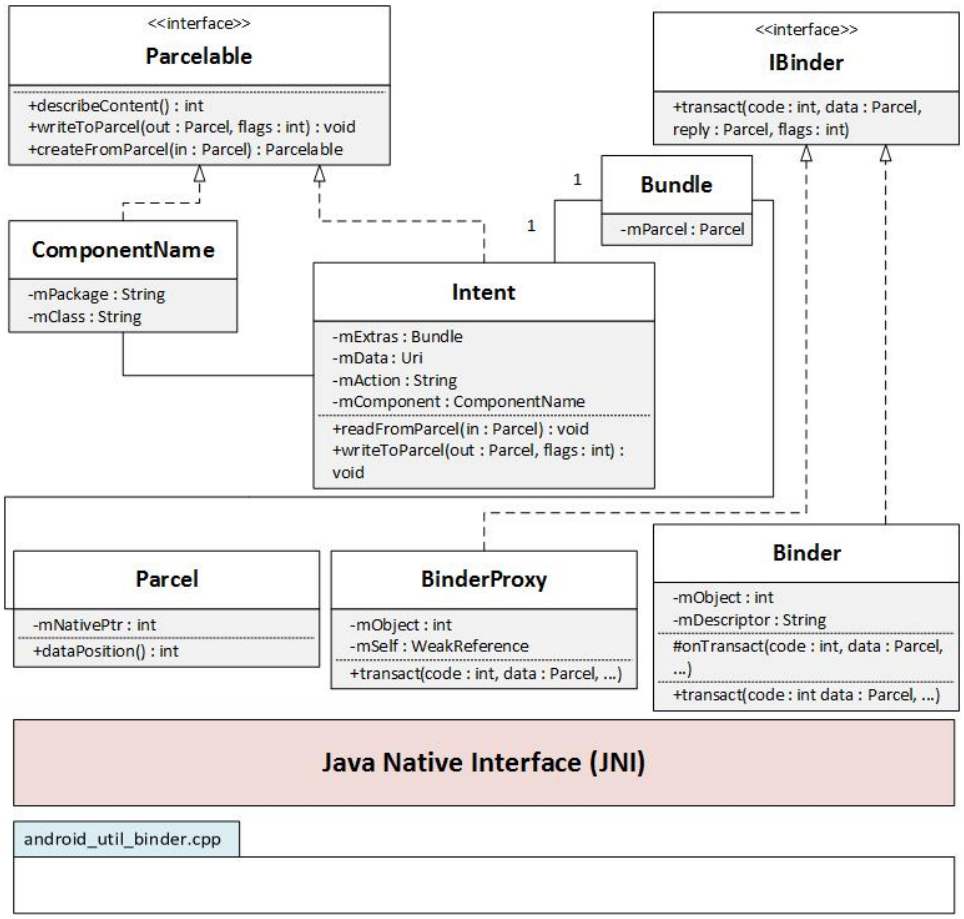

Figure 3.5: UML Diagram of the Main Components of Binder APIs

The Figure 3.5 shows an high-level summary of the Java part of Binder framework. The main components are `Binder`, `Intent` and `Parcel`.

All classes related to `Binder` are simply the Java correspondance to underlying C++ objects. Indeed, the memory sharing facilities of Binder can not be used by Java [18], only native C++ libraries can have access to shared object representations in memory.

`Intent` contains a specific field (`mAction`) that refers to a method defined in the programming interface that both the client and service agree upon in order to communicate each other. Intent utilizes `Bundle` container to collect heterogeneous values. It is organized as a key-value pairs table for data identification between processes.

```
// Client
intent.putExtra("key", new MyObj(...));
// Server
Bundle data = getIntent().getExtras();
MyObj mObj = data.getParcelable("key");
```

`Parcel` is the java interface for data serializazion. A process cannot access the memory of another one, so the client needs a mechanism to act a "serialization" of its objects into raw data that Android can easily move to the memory space of the server, which on its side has perform the dual operation called "deserialization". Android have standard mechanism to transfer basic types and provides a simple interface (`Parcelable`) that an object must implement to be sent to another application. They constitute a high-performance IPC transport and use an extremely efficiency (but low-level) protocol for objects to write or read.

The module `android_util_binder.cpp` belongs to JNI and maps JAVA APIs to the corresponding C++ functions.

3.8 C++ Middleware

The Figure 3.6 shows an high-level summary of the C++ middleware. This part is the core of the Binder framework and provides:

- an implementation of stub (`Binder.h(.cpp)`) and proxy `BpBinder.h (.cpp)`), both derived from the common interface (`IBinder.h`).
- an interface (`Parcel.h(.cpp)`) responsible for objects marshalling and unmarshalling.

  This module provides several methods to read and write basic type elements, objects, binder, file descriptor, and so on. It handles an array (`uint8_t* mData`) of variable size and performs the conversion between high-level objects to raw data and viceversa.

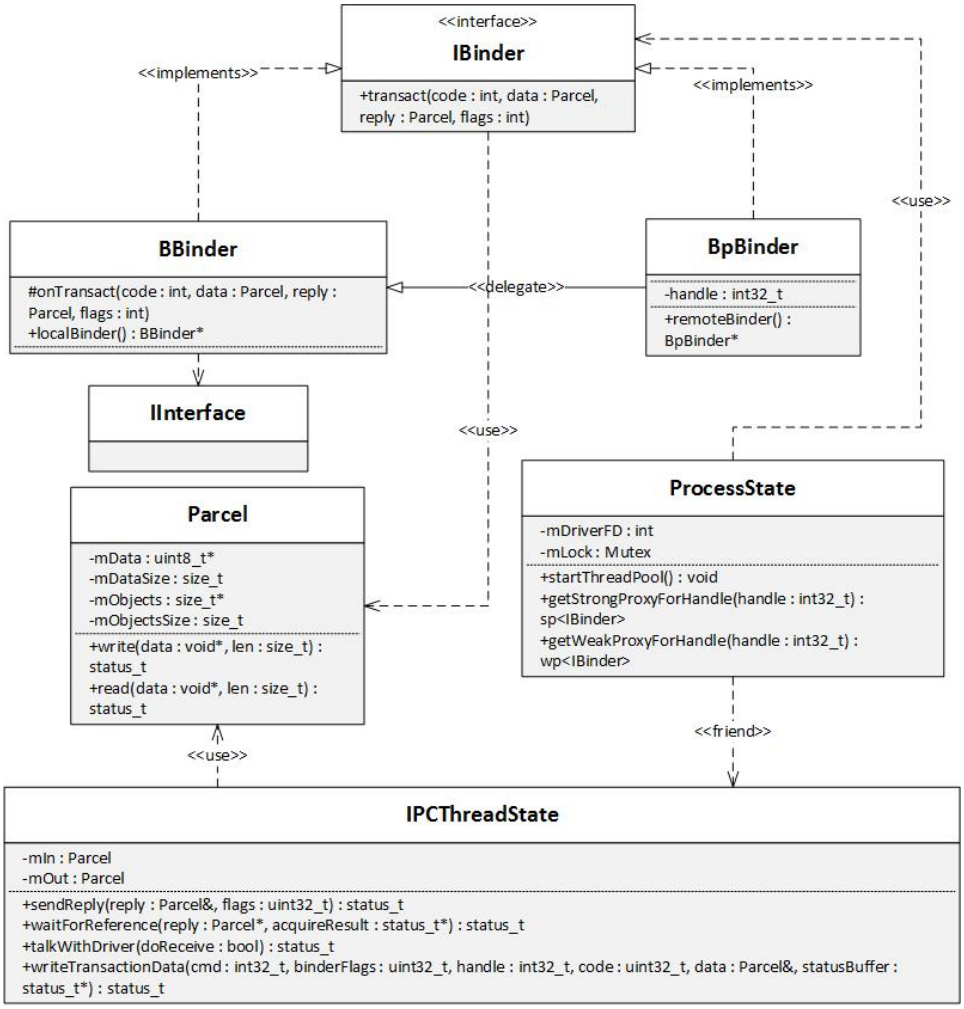

Figure 3.6: UML Diagram of the Main Components of Binder Framework

These components have a direct correspondence with the Java ones.

- an efficient mechanism to implement a pool of threads to serve client requests (`ProcessState.h`, `IPCThreadState.h`).

  The former module is responsible for:

  - opening the device driver;
  - mapping the binder memory;
  - providing a chunk of virtual address space to send and receive transaction data;
  - initializing and handling the pool of threads in order to speed-up response time.

The latter takes care of:

- writing and sending data;
- waiting for responses;
- interacting with the driver.

After the end of a transaction, the thread can be re-used.

These components are at the lowest level of user space.

3.8.1 Remote Method Invocation

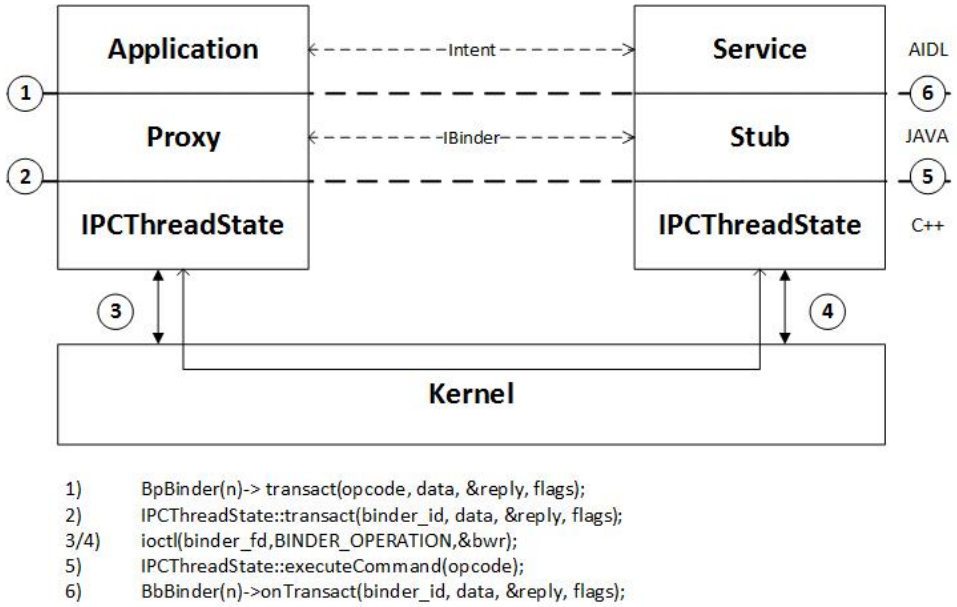

Figure 3.7: Binder Stack for Remote Method Invocation

The Binder Interface - automatically generated by the `.aidl` file - contains the key APIs `IBinder.transact()` and `Binder.onTransact()`, respectively for the client and for the server, which are directly responsible for transaction data execution.

The former serves to invoke a service, the latter makes the service waiting for requests. The transaction is **synchronous**, such that a call to `transact()` does not return until the target has returned from `Binder.onTransact()`. This is the expected behavior when calling an object that exists in the local process, and the underlying inter-process communication mechanism ensures that the same semantics stands also among processes.

The pairs of function [`IBinder.transact()` - `Binder.onTransact()`] and the `Parcel` class exist both in Java and in the middleware. The only

relevant difference is that on the server side there is a pool of threads that are blocked on the `onTransact()` method, hence the Binder process has to manage awakenings.

3.9 Kernel Module

The Kernel module constitutes the lowest level part of the Binder Framework and it is the only component that have the awareness of the whole set of binders in the systems [3].

Working inside the kernel, the verse of both write and read operation are reversed. It means that when user calls a write operation, the driver has to get data from user space; on the other way, when user calls a read operation, the driver has to put data to user space.

The binder module ("`/dev/binder`") exploits the miscellaneous device library of the Linux kernel in order to implement the basic file communication system calls (`poll`, `ioctl`, `mmap`, `open`, `flush`, `release`).

3.9.1 Binder Protocol

At the lowest level the Framework utilizes a `ioctl()`-based low-level protocol to communicate with the Binder driver: :

```
ioctl(int fd, unsigned int cmd, unsigned long arg)
```

To be more precise, the driver implements the *unlocked* version of the system call: `long ioctl(struct file *file, unsigned int cmd, unsigned long arg)`), where the first argument points to the file structure that represents the device. The operating system takes care of translation between file index and descriptor. The arguments are:

- The file descriptor of ("`/dev/binder`"), which represents the device;
- The command passed to the kernel: Android Binder supports five commands:
    1. **BINDER_WRITE_READ**, deeply used for data transaction. It's the unique command that directly respond to an user request;
    2. **BINDER_SET_MAX_THREADS**, used by the frameworks to set the max number of threads;
    3. **BINDER_SET_CONTEXT_MGR**, used by the service manager during the initialization process;

[3] Processes knows only their own binders and those requested to and provided by the Service Manager.

4. **BINDER_THREAD_EXIT**, called by the thread destructor to warns the kernel to remove the data structures related to the thread itself;
5. **BINDER_VERSION**, used to know the current version of the protocol when the driver is opened.

- The pointer to a user buffer that is, for each command:

1. A `binder_write_read` data structure;
2. The maximum number of threads;
3. A dummy pointer;
4. A NULL pointer;
5. The number of version.

The commands listed above correspond to very simple operation except for the first case - **BINDER_WRITE_READ** - which is the most important command and is utilized both for data exchange and for remote method invocation [4].

The protocol used by this command is structured in a quite complex way that hereby we set about discussing. The third argument of the system call points to a data structure containing two references to buffers: one for reading and one for writing.

```
struct binder_write_read {
  signed long   write_size;     // bytes to write
  signed long   write_consumed; // bytes consumed by driver
  unsigned long write_buffer;   // address of write buffer
  signed long   read_size;      // bytes to read
  signed long   read_consumed;  // bytes consumed by driver
  unsigned long read_buffer;    // address of read buffer
};
```

These two buffer contains a sequence of <commands-arguments_list> pairs that the driver parses and executes. The command are coded using the prefix BC (Binder Command) in the write buffer and BR (Binder Return) in read one (Figure 3.8). A positive value of the size variable indicates to the driver to start reading/writing procedures. Among all Binder protocol subcommands, which will be described in the following Chapter, the most important are BC(BR)_TRANSACTION and BC(BR)_REPLY because they are related with data transactions.

[4] At this level these two operation are indistinguishable: a RMI is basically a transfer of a code that identify the function and of a data buffer containg arguments.

```
void __user *ubuf = (void __user *)arg;
...
switch (cmd) {
  case BINDER_WRITE_READ: {
    struct binder_write_read bwr;
    copy_from_user(&bwr, ubuf, sizeof(bwr));
    ...
    if (bwr.write_size > 0) {
      ret = binder_thread_write(...);
    }
    if (bwr.read_size > 0) {
      ret = binder_thread_read(...);
     }
    copy_to_user(ubuf, &bwr, sizeof(bwr));
    break;
  }
}
```

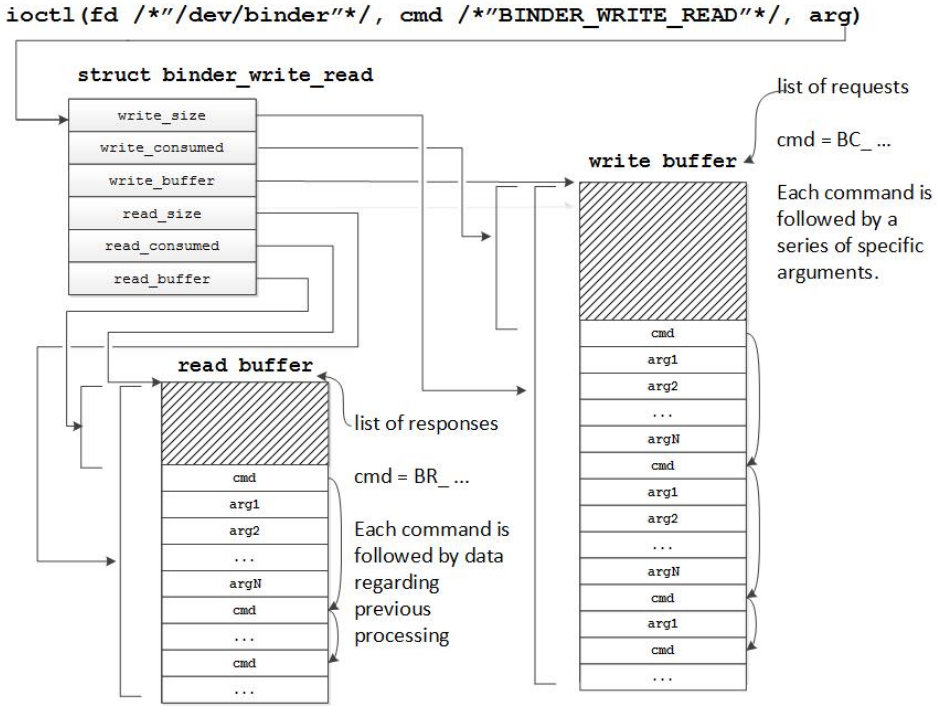

Figure 3.8: Low-Level Execution of Binder Transaction

Chapter 4

Implementation Details

In this Chapter we focus on some fundamental implementation details and on the low-level behavior of the system.

We deepen the mechanisms of Services Registration and Lookup and Remote Method Invocation through Binder, afterwards we describe the main components of the Kernel Module.

4.1 Service Registration and Lookup

Binder IPC can work only if these assumption are verified:

1. The server has registered the service;
2. The client knows the *name* of the service.

ServiceManager handles the registration process providing the method:

```
public void addService(String name, IBinder service, ... );
```

that publishes the `name` and the `service` [1]. A client can *bind*[2] itself with the server using the `getService()` method:

```
public IBinder getService(String name);
```

that takes the name of the service as argument and returns the proper binder interface[3].

The client process has to open the driver and to map some virtual addresses (8KB - 1MB of memory) for data transaction. When the client obtains the requested binder, it only has to convert it in a service interface:

[1] It's the interface - derived from `IBinder` that is automatically generated by AIDL.
[2] The client must have the right permissions to use the service.
[3] It is created once at initialization time

```
IRemoteService mIRemoteService;
mIRemoteService = IRemoteService.Stub.asInterface(service);
```

The method can be invoked normally by Java code. The framework takes care of translate the method invocation in a binder transaction.

```
virtual status_t addService(const String16& name,
    const sp<IBinder>& service, ...) {
  Parcel data, reply;
  data.writeInterfaceToken(IServiceManager::
     getInterfaceDescriptor());
  data.writeString16(name);
  data.writeStrongBinder(service);
  ...
  status_t err = remote()->transact(ADD_SERVICE_TRANSACTION
     , data, &reply);
  return err == NO_ERROR ? reply.readExceptionCode() : err;
}
```

```
virtual sp<IBinder> getService (const String16& name) const
    {
  ...
  sp<IBinder> svc = checkService(name);
  if (svc != NULL) return svc;
  ...
  return NULL;
}

virtual sp<IBinder> checkService( const String16& name)
   const {
  Parcel data, reply;
  data.writeInterfaceToken(IServiceManager::
     getInterfaceDescriptor());
  data.writeString16(name);
  remote()->transact(CHECK_SERVICE_TRANSACTION, data, &
     reply);
  return reply.readStrongBinder();
}
```

The code of the native methods gives us an opportunity for some observations that will become more clear when we explain how driver manage binder object:

- the target of transaction is the `ServiceManager`, the only binder whose token (0) is known by all the system;
- the registration of a service consists in writing the corresponding binder in a parcel and send it into the driver;

- the lookup consists in reading the binder of a service of which we know the name.

4.2 The AIDL Interface

Once an Activity owns a binder of a service, it can invoke functions and methods as it was local. Indeed, both the proxy and the stub class extends `Binder` class [4], which hides all details on the service but the provided methods. The steps an application should make to call the remote interface are very simple [5]:

1. declaration of a variable of the interface type defined in a `.aidl` file;
2. binding to the service calling `Context.bindService()`;
3. at the end, disconnection calling `Context.unbindService()`.

4.3 Proxy and Stub

A brief analysis of the native part of Binder cannot begin from anywhere but `IBinder.h`, which is the common interface between proxy and stub. Its main method are:

- `localBinder()`, which returns a reference to the local object;
- `remoteBinder()`, which returns a reference to the remote object;
- `transact()`, which is responsible to forward the transaction to the driver.

```
class IBinder : public virtual RefBase {
public:
  ...
  virtual status_t transact (uint32_t code,
                    const Parcel& data,
                    Parcel* reply,
                    uint32_t flags = 0) = 0;
  ...
  virtual BBinder* localBinder();
  virtual BpBinder* remoteBinder();
};
```

The Table 4.1 contains main difference between the implementation in `BBinder` (the stub) and `BpBinder` (the proxy).

[4] `Binder` class implements `IBinder` interface

[5] Generally these operations are supported by `ServiceConnection`: an interface for monitoring the state of an application service [8]

Interface	Stub	Proxy
`localBinder()`	`{return this;}`	Not implemented
`remoteBinder()`	Not implemented	`{return this;}`
`transact()`	`{err = onTransact(code, data, reply, flags);}`	`IPCThreadState:: self()->transact (mHandle, code, data, reply, flags);`

Table 4.1: Functions Comparison between Proxy and Stub

4.4 Kernel Module Components

In order to fully understand low-level Binder Protocol, we need to list and to describe the key data structures used by the driver.

4.4.1 Nodes and References

The driver wraps all the useful information concerning Binder objects in a `binder_node`. The main fields are:

- a pointer to the object in the owner process user space, so it identifies uniquely the object itself;
- the process descriptor;
- the list of reference to the object.

There is a special instance with identifier **0** created at installation time and reachable by all the processes in the system[6].

```
struct binder_node {
   ...
   struct binder_proc* proc;
   struct hlist_head refs;
   ...
   void __user *ptr;
   ...
};
```

The driver also manages the reference counting to these binder objects using a thinner data structure containing:

[6]The other binders' identifier is process-dependent. The driver takes care of translation (i.e. the same binder may have different identifiers in different processes) during binder transactions.

- a pointer to the descriptor of binder process;
- a reference to the binder object;
- a unique token;
- a counter for "strong" references;
- a counter for "weak" references.

The driver stores the references in a Red-Black tree.

```
struct binder_ref {
    ...
    struct binder_proc *proc;
    struct binder_node *node;
    uint32_t desc;
    int strong;
    int weak;
    ...
}
```

While there is only one node per binder [7], there can be lots of references per binder [8]. A binder object exists in its process space till it has at least one strong reference.

4.4.2 Processes and Threads

The binder driver holds the list of processes opened by the driver itself. Each process has its own descriptor that is initialized when it opens the driver. It contains all data structures associated with the process itself:

- The pool of threads that the process manage in order to perform pending transactions;
- The set of binder owned by the process;
- The set of references to binder objects;
- The stack of pending transaction;
- The wait queue in which idle threads are blocked;
- Other stuff related to task, files and memory management.

[7] That is, a `binder_node` is a binder instance.

[8] That is, one for each process that has required that particular binder. All this references points, inside the kernel, to the same binder object

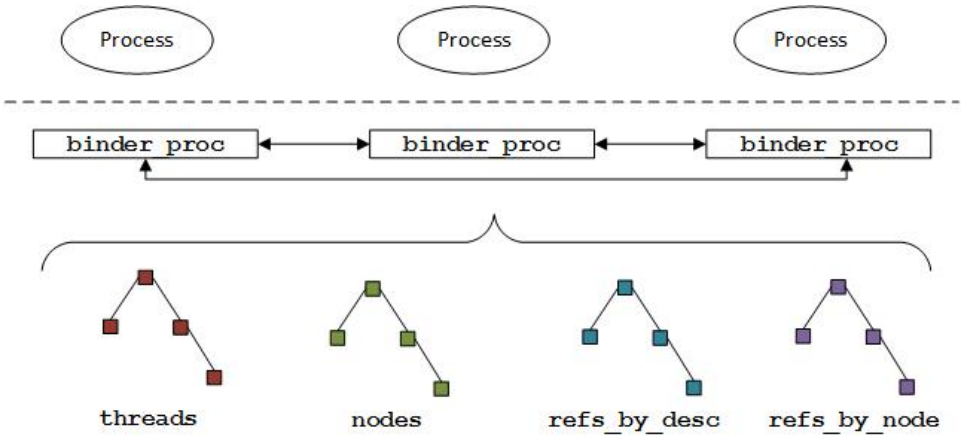

Figure 4.1: Data Structure of Binder Process Descriptor

In the kernel space, local binders of a process are referenced directly by `binder_node` pointers sorted in nodes r-b tree in the particular `binder_proc` , while remote binders are primarily referenced indirectly via descriptors (it is just driver's word for the handle appearing in `flat_binder_object`) sorted into `refs_by_desc` and `refs_by_node` trees in `binder_proc` [17].

```
static HLIST_HEAD(binder_procs);

struct binder_proc {
  struct hlist_node proc_node;
  struct rb_root threads;
  struct rb_root nodes;
  struct rb_root refs_by_desc;
  struct rb_root refs_by_node;
  ...
  struct list_head todo;
   ...
};
```

The Binder exploits a pool of threads for each binder process, in order to rapidly handle user request. Each thread has its own descriptor (`binder_thread`) containing:

- The thread identifier;
- The reference to the process;
- The stack of binder transaction to handle;
- The wait queue.

The driver provides a simple mechanism for a thread to register and delete itself as a "service provider".

```
struct binder_thread {
   ...
   int pid;
   struct binder_proc *proc;
   struct binder_transaction *transaction_stack;
   wait_queue_head_t wait;
   ...
};
```

There is a maximum number of thread which is set by the user-level when the binder is initialized. All binder threads in a pool are blocked in the `proc->wait` queue waiting that the driver dispatches a transaction to one of them. When the execution is finished, the thread returns in the queue.

4.4.3 Transaction

The driver exchanges data between processes using binder a list of commands. They are split into "*command*" if the direction is from the user to the kernel (write operations) and "*return*" otherwise (read operations). The paradigm is client-server (a simple request and reply [9]). Each `transact()` call will produce a `binder_transaction` object in the kernel, the object is sent to the target process passing from the `todo` queue of handler thread, at the end the thread waiting in `onTransact()` call is awaened and it finishes the transaction.

```
struct binder_transaction {
  ...
  struct binder_work work;
  struct binder_thread *from;
  struct binder_transaction *from_parent;
  struct binder_proc *to_proc;
  struct binder_thread *to_thread;
  struct binder_transaction *to_parent;
  unsigned need_reply:1;
  struct binder_buffer *buffer;
  ...
};
```

The driver holds all information concerning transaction in a data structure, particularly:

[9] If needed.

- The type of work to do (e.g. a transaction, a completed transaction, an operation on a binder, a dead binder notification, ...);
- The thread which requested the transaction[10];
- The last `binder_transaction` for the client (`from_parent`) and for the server (`from_parent`) [11];
- The target process;
- The target thread;
- The last `binder_transaction` for the server;
- The data buffer [12].

4.4.4 Buffer

Inter process communication allows to move data from one process' address space to another one. This operation usually need two copies [13].

Using `binder_buffer` data structure, the kernel performs only a single copy to improve the efficiency of the Android Binder. It is possible because the kernel address space of binder process is mapped to user space through the `mmap()` system call, hence it can directly access the content without an extra copy. The descriptor contains:

- The transaction that uses this buffer;
- The target object;
- The size in bytes of the data;
- The buffer (the data is in the form of Parcel).

```
struct binder_buffer {
  ...
  struct binder_transaction *transaction;
  struct binder_node *target_node;
  size_t data_size;
  ...
  uint8_t data[0];
};
```

[10] If BC_TRANSACTION, then the client thread, if it is BC_REPLY, then the server thread.
[11] Binder Transactions are organized in a stack
[12] The `binder_buffer` data structure is needed to wrap the memory buffer with management information (linked transaction, flags and so on ...).
[13] Process A $\Longrightarrow$ Kernel, Kernel $\Longrightarrow$ Process B

4.4.5 Binder Object

A Binder Object can be shared among different processes. This fact permits to register services [14]) and to receive remote reference of an object.

Instead to send the whole interface, the driver uses a flattened representation of a Binder object. It takes care of re-writing the structure type and data as it moves between processes to maintain token uniqueness and to manage the transition from local to remote and viceversa.

The flattened data structure is only 16 bytes long and holds only:

- the type of binder (weak/strong, local/remote, file descriptor[15]);
- a 32-bit variable for flags;
- the id, which is a pointer if the binder is local (i.e. if it was created by the process) or an integer if the binder is remote;
- the cookie, an object reference used for identification of death receptors.

```
struct flat_binder_object {
  unsigned long  type;
  unsigned long  flags;
  union {
    void *binder;
    signed long handle;
  };
  void *cookie;
};
```

4.4.6 Binder Transaction Data

The core functionality of Binder is data exchange between applications. Among the big set of Binder commands, those really important takes care of data transfer. They uses the descriptor shown below in order to handle user data [16]. It is used for both write and read operation: the driver takes care of appropriate translation (e.g. memory addresses, binder identifier and cookie).

[14]It's sufficient that a Binder pass through the driver to create the node and the first reference.

[15]Binder objects are used also to pass files. The flattened representation contains the file descriptor.

[16]`target` field (and the following `cookie`) are only used for BC_TRANSACTION and BR_TRANSACTION, i.e the commands related to a request of something. They identify the target and contents of the transaction.

```
struct binder_transaction_data {
  union {
    size_t handle; // command transaction
    void *ptr;     // return transaction
  } target;
  void  *cookie;       // target object cookie
  unsigned int  code;  // transaction command
  unsigned int  flags; // transaction flags
  pid_t sender_pid;    // pid of sending process
  uid_t sender_euid;   // euid of sending process
  size_t data_size;    // number of bytes of data
  size_t offsets_size; // number of bytes of offsets
  union {
    struct {
      const void *buffer;  // transaction data
      const void *offsets; // offsets from buffer
    } ptr;                 // to flat_binder_object
    uint8_t buf[8];
  } data;
};
```

Chapter 5

Communication Protocol

In this Chapter we examine in depth communication protocol between user and kernel, particularly we describe `BINDER_WRITE_READ` transaction. Our analysis runs through all levels, from user application to system calls execution, but focuses on the low-level protocol.

5.1 Binder Driver Commands

BINDER_WRITE_READ command encapsulates a very large set of sub-commands that can be divided firstly in two categoires: write with "**BC**" as prefix (*Binder Command*) and read with "**BR**" as prefix (*Binder Return*). The values are composed using conventional macros for `ioctl` command number:

- `IO(int type, int number)` used for a simple `ioctl` that sends nothing but the type and number, and receives back nothing but an integer;
- `_IOR(int type, int number, data_type)` used for an `ioctl` that reads data from the device driver. The driver will be allowed to return to the user the size in bytes of the data structure ;
- `_IOW(int type, int number, data_type)` similar to `_IOR`, but used to write data to the driver;

where:

- `type` is an 8-bit integer selected to be specific to the device driver. It should be chosen so as not to conflict with other drivers that might be listening to the same file descriptor.
- `number` is an 8-bit integer command number. Within a driver, distinct numbers should be chosen for each different kind of command that the driver services.

- `data_type` is the name of a type used to compute how many bytes are exchanged between the client and the driver. This argument is, for example, the name of a structure.

There are some commands which we could define "dual", because they are used both in writing and in reading operations. It means that the direction of the command is from a process to another process or, better, that for each writing operation by a process (the client) there is another process (the server) waiting on a reading operation.

In the Table 5.1 and 5.2 we have summarized the commands grouped by types that share data format.

A deep analysis of the execution of each single commands is outside of the scope of this thesis. Moreover, we believe that a survey on the flow of commands generated by a user request (i.e. an invocation of native method `transact(...)` is more interesting and helpful to reach our goals. A carfeul reading of source code ([10] and [9]) should make clear the behavior of at least simple commands.

5.2 Binder Communication Protocol for Data Transaction

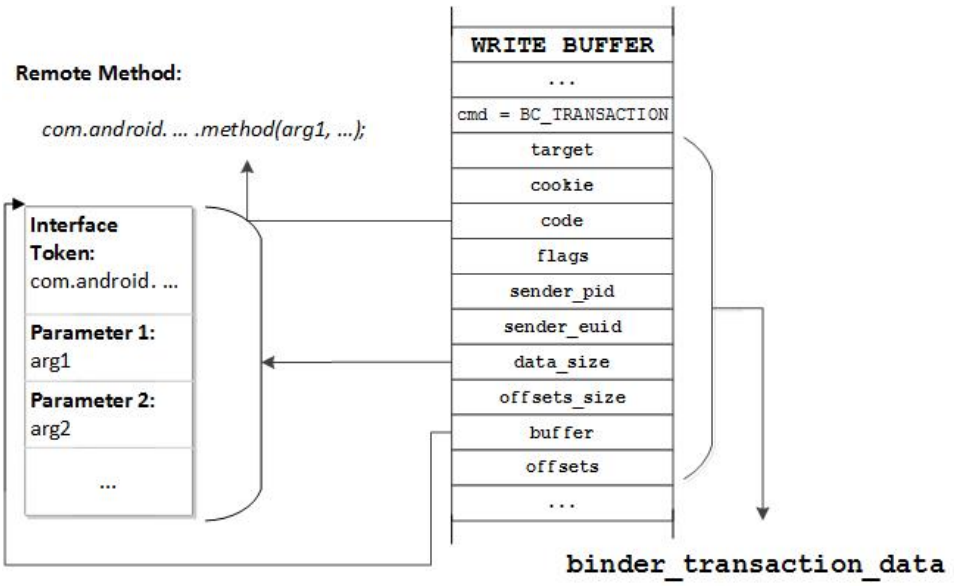

Figure 5.1: Classical Format a Binder Data Transaction

For the sake of simplicity, we concentrate our analysis on the data transaction commands. We made this choice for mainly two reasons:

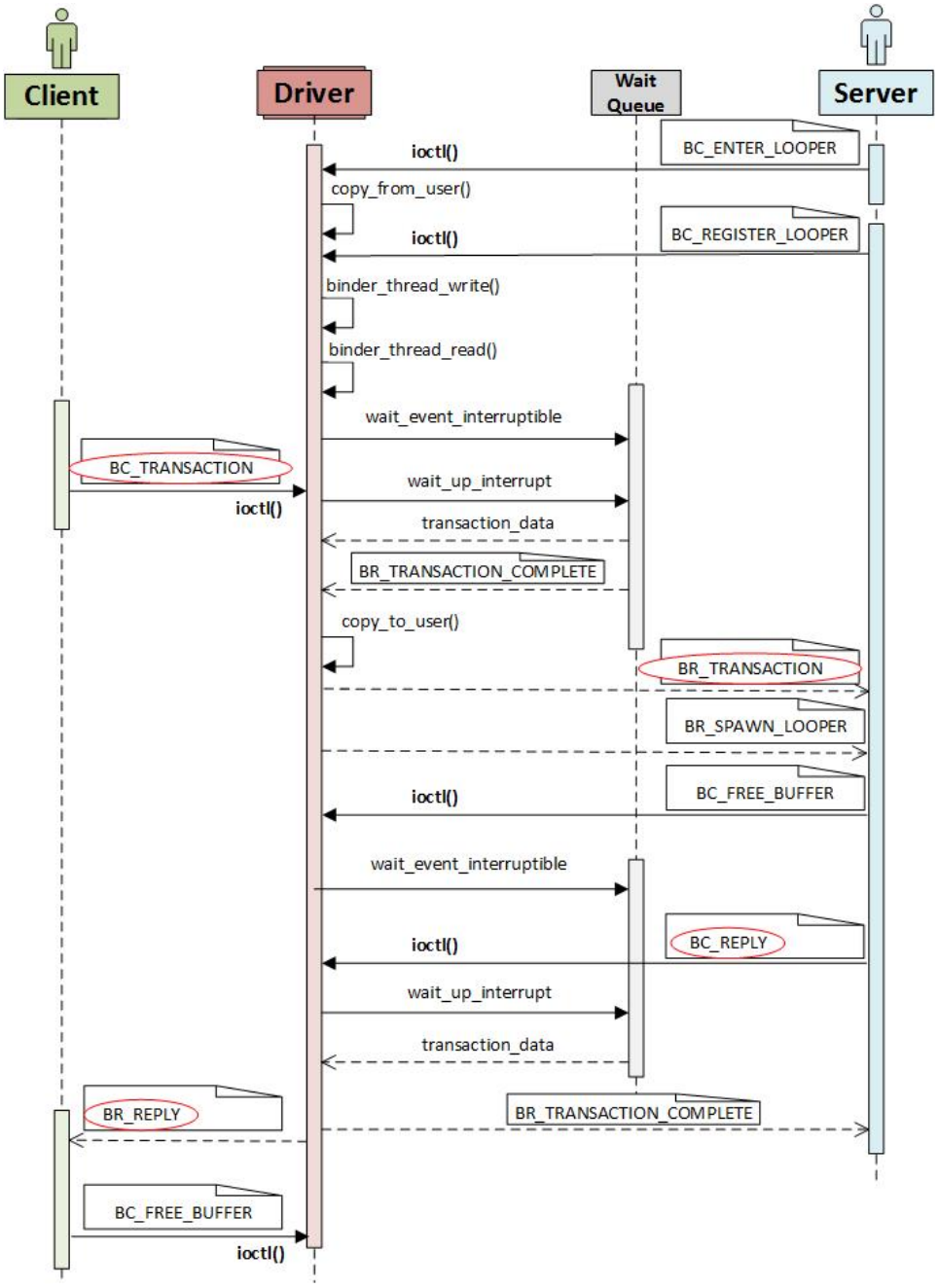

Figure 5.2: Sequence Diagram of a Method Invocation

1. They are the only commands to handle user buffer;

2. They are launched by application request, instead of reference counting or thread management commands, which are directly handled by the framework;

The diagram in the Figure 5.2 represents in details the sequence of Binder driver calls behind a simple method invocation (or data exchange) between two processes. There is not a bijective correspondence between the diagram and the effective `ioctl` invocations because the communication protocol allows **BINDER_WRITE_READ** command to embed more than one subcommands.

If we ignore the commands related to thread management by the server, to the synchronization and to the copy of data we obtain the skeleton (circled in red) of a classic two way Binder transaction.

5.2.1 Command Protocol

The Command Protocol is managed by `binder_transaction(...)` method, which:

BC_TRANSACTION

- searches the reference to the target binder using its token (if the target is the *Service Manager*, it uses the default one) ;
- recovers the node and process descriptors using the reference;

BC_REPLY

- recovers the transaction from the stack of the thread;
- sets the `target_proc`, the `target_thread` using transaction descriptor;

then the body of the method is common:

- Setting of the `target_list` (the schedule of the work to do[1]) and the `target_wait` () with those of `target_thread` - if exists - or of `target_proc` [2];
- Memory allocation for `binder_transaction` and `binder_work` [3] data structures;
- Creation of a Binder buffer in the memory space of target process;

[1]For `binder_thread_read(...)`

[2]Generally, a target thread exists if the process offer a service, so implement a pool of thread to speed-up providing.

[3]Serve per la binder_thred_read

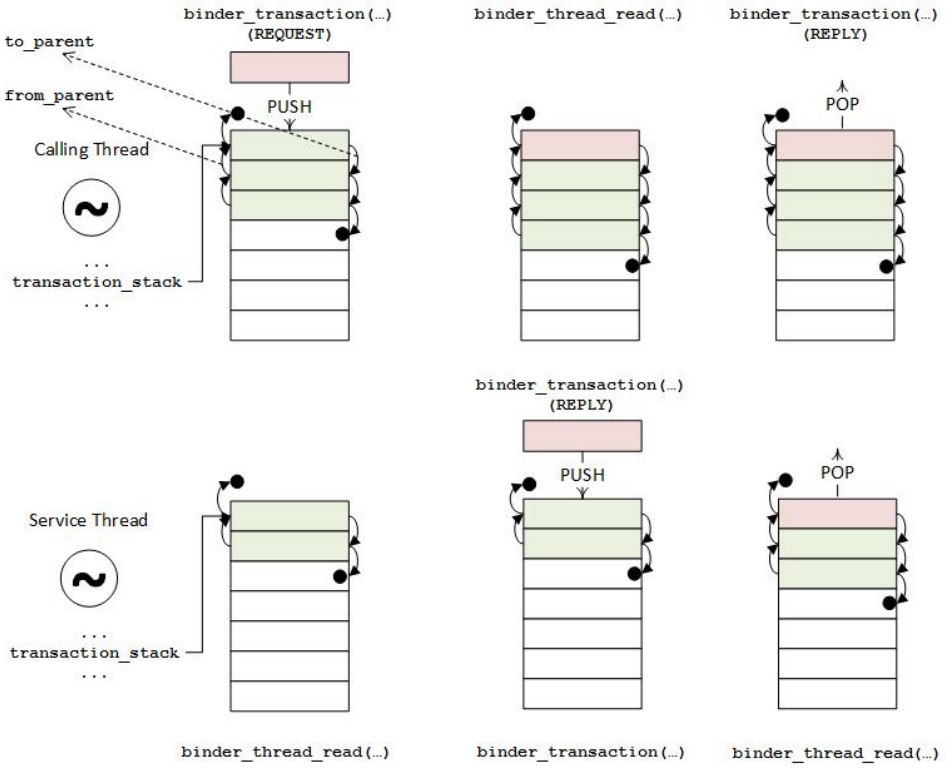

Figure 5.3: Evolution of Thread Stack during data transaction

- Initialization of `binder_transaction` using user data[4];
- Copy of user data (both the data buffer and the offsets one) in the allocated buffer;
- Loop on the whole set of offsets:
  - Recover the flatten representation of the binder object;
  - Update some fields of flat binder object to make it consistent the target process;

BINDER_WEAK_BINDER

- Take the node and the reference to the binder object;
- Set the handle with the reference's descriptor;
- Increment the reference counting;

BINDER_WEAK_HANDLE

- Take the reference using the handle of the binder;
- If exists the binder, then it's initialized and incremented, otherwise it's created;

- Check the type of transaction:

REPLY

- Pop from the stack

TWO_WAY

- Push in the stack

ONE_WAY

- Set one way flag;

- Adding of `binder_transaction` in the `target_list` with BINDER_WORK_TRANSACTION as work type;
- Waiting on the `target_wait` queue[5].

5.2.2 Return Protocol

The return protocol is managed directly inside `binder_thread_read(...)`, which is quite different from its dual because it has to manage synchronization. Indeed, an application who invokes Binder driver in read mode must block itself until data are not ready. It means that read operation can temporally precede the write one.

[4]User data are in the `binder_transaction_data` data structure, which is passed as argument to this function, and is recovered by the buffer of user command.

[5]It will be BR_TRANSACTION and BR_REPLY commands with the appropriate target to wakeup the thread.

```
// Body of binder_thread_read: 1st part.
binder_unlock(__func__);
...
ret = wait_event_interruptible(thread->wait,
    binder_has_thread_work(thread));
...
binder_lock(__func__);
...
```

`binder_thread_read(...)`distinguishes the operations to execute using the work type of the transaction in the stack instead of the user commands. After the awakening of a thread, the method recovers the list of pending work and loop on the whole collection:

```
// Body of binder_thread_read: 2nd part.
while (1) {
  uint32_t cmd;
  struct binder_transaction_data tr;
  struct binder_work *w;
  struct binder_transaction *t = NULL;
  if (!list_empty(&thread->todo))
    w = list_first_entry(&thread->todo, struct binder_work,
        entry);
  ...
  switch (w->type) {
    case BINDER_WORK_TRANSACTION: {
      t = container_of(w, struct binder_transaction, work);
    } break;
    case BINDER_WORK_TRANSACTION_COMPLETE: {
      cmd = BR_TRANSACTION_COMPLETE;
      put_user(cmd, (uint32_t __user *)ptr))
      ...
    } break;
  }
}
```

As in the command protocol **BC_TRANSACTION** and **BC_REPLY** is the main, in the return protocol the **BINDER_WORK_TRANSACTION** is by far the most important because takes care of the second part of data transaction, which consists in the building of a `binder_transaction_data` in the read buffer. [6], which basically consists in creating and initializing a `binder_transaction_data` and in copying it into the read buffer of the user.

[6]Remember by the sequence diagram that a data transfer requires a system call on both sides, the former in write mode, the latter in read mode.

```
// Body of binder_thread_read: 3rd part.
if (t->buffer->target_node) {
  struct binder_node *target_node = t->buffer->target_node;
  tr.target.ptr = target_node->ptr;
  tr.cookie =  target_node->cookie;
  ...
  cmd = BR_TRANSACTION;
} else {
  tr.target.ptr = NULL;
  tr.cookie = NULL;
  cmd = BR_REPLY;
}
tr.code = t->code;
tr.flags = t->flags;
...
tr.data_size = t->buffer->data_size;
tr.offsets_size = t->buffer->offsets_size;
tr.data.ptr.buffer = (void *)t->buffer->data + proc->
    user_buffer_offset;
tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer
    ->data_size, sizeof(void *));
...
copy_to_user(ptr, &tr, sizeof(tr));
```

5.3 Binder Object Exchange

One key function of the driver is the transfer of Binder Object between processes. It uses `flat_binder_object` representation in order to save memory. The driver has to manage the consistency of Binder objects during the change of memory space. In short - local binders are represented directly by pointers to the user space objects, while remote binders are referred by handles. In the kernel driver, there is a system-wide unique representation of each single binder by `binder_node` structure instance. It consists in an incremental counter for each process. It means that in different processes a binder could have different handle, but the driver takes care of translation and guarantees the correspondence.

5.4 Internal Bug

While studying the internals of Binder, we discovered a problem in its current implementation. In particular, we found that, under specific circumstances, the kernel module does not properly sanitize transaction data. This allows a malicious application to send invalid transaction data to any process reachable by using Binder, making such process to crash. In addition,

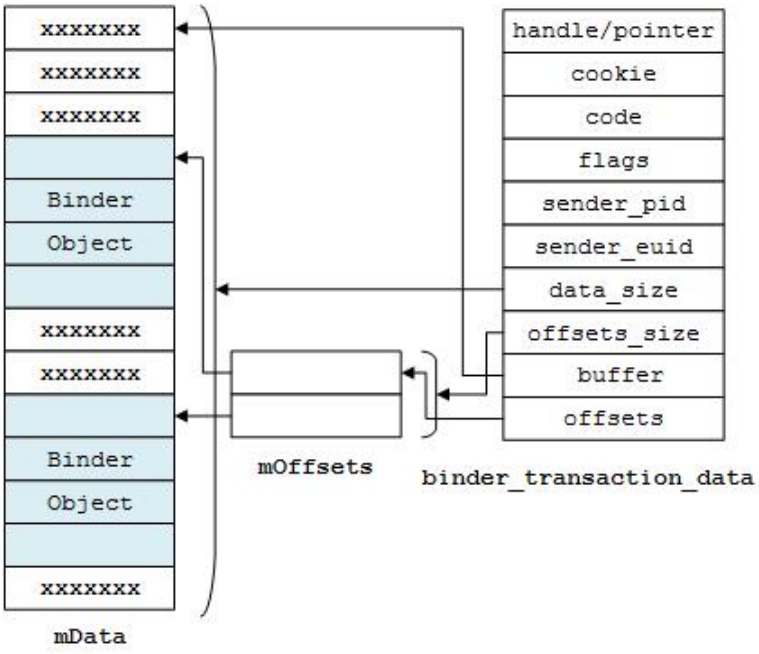

Figure 5.4: Binder Encapsulation in Parcel Data Buffer

even though we were not able to produce this behavior in our experiments, the same bug could also lead to arbitrary code execution in the context of the reached process.

We communicated this problem to the Android security team, that acknowledged it and released a specific patch to the Binder kernel module. However, at the moment of the writing of this thesis, the latest release of Android includes an unpatched version of the Binder kernel module. For this reason, at this time, we do not disclose further details about this problem.

TYPE	COMMAND LISTS	DATA FORMAT (Read from the user buffer)
Reference Counting	BC_INCREFS BC_ACQUIRE BC_RELEASE BC_DECREFS	|------cmd-----| |----target----|
Reference Counting	BC_ACQUIRE_DONE BC_INCREFS_DONE	|------cmd-----| |---node_ptr---| |----cookie----|
Buffer Management	BC_FREE_BUFFER	|------cmd-----| |---data_ptr---|
Data Transaction	BC_TRANSACTION BC_REPLY	|------cmd-----| |----target----| |----cookie----| |-----code-----| |--sender_pid--| |--sender_euid-| |---data_size--| |-offsets_size-| |-data_buffer--| |offsets_buffer|
Thread management	BC_ENTER_LOOPER BC_REGISTER_LOOPER BC_EXIT_LOOPER	|-----cmd------|
Death Notification	BC_REQUEST_ DEATH_NOTIFICATION BC_DEAD_BINDER_DONE	|------cmd-----| |----target----| |----cookie----|
Death Notification	BC_CLEAR_ DEATH_NOTIFICATION	|-----cmd------| |----cookie----|

Table 5.1: Binder write commands with data formats

WORK TYPE	**COMMAND LISTS**	**DATA FORMAT** (Write to user buffer)
NODE	BR_INCREFS BR_ACQUIRE BR_RELEASE BR_DECREFS	|-----cmd------| |-----ptr------| |----cookie----|
DEAD_BINDER DEAD_BINDER_ AND_CLEAR CLEAR_DEATH_ NOTIFICATION TRANSACTION_ COMPLETE	 BR_CLEAR_DEATH_ NOTIFICATION_DONE BR_DEAD_BINDER BR_TRANSACTION_ COMPLETE	 |-----cmd------|
TRANSACTION	BR_TRANSACTION BR_REPLY	|-----cmd------| |----target----| |----cookie----| |-----code-----| |--sender_pid--| |--sender_euid-| |---data_size--| |-offsets_size-| |-data_buffer--| |offsets_buffer|

Table 5.2: Binder read commands with data formats

Part II

Presentation of Jarvis

Chapter 6

Description

In the first part we have briefly introduced some topics concerning Android Operating System and Binder IPC that we retained fundamental in order to understand the remaining portion of this dissertation. We took for granted the knowledge of Linux kernel.

In this Chapter we describe the tool we have implemented: the main features and capabilities, the high-level schema and the organization in modules.

6.1 General Information

Jarvis is a tool committed to bridge the semantic gap between high-level Android APIs and low-level System Calls by means of tracking, filtering and logging system calls in a new, Android-specific way. The purpose of Jarvis are:

1. mapping APIs and system calls, in order to understand the sequence of kernel call associated to a particular high-level function, which imply the implementation of a filter mechanism.
2. rebuilding high-level behavior from a System Calls' log;

The main components are:

- A kernel module that takes care of sniffing, filtering, logging and storing of a collection of the most invoked system calls;
- A simple Android application to exploit some high-level classes, which are the `Classloader` and the `Parcel`, to provide elaborated data;
- A tool to automatically invoke a list of APIs, in order to reconstruct their "signature";

- Some configuration and parsing scripts.

The tool is available at [20] and is structured as in Figure 6.1.

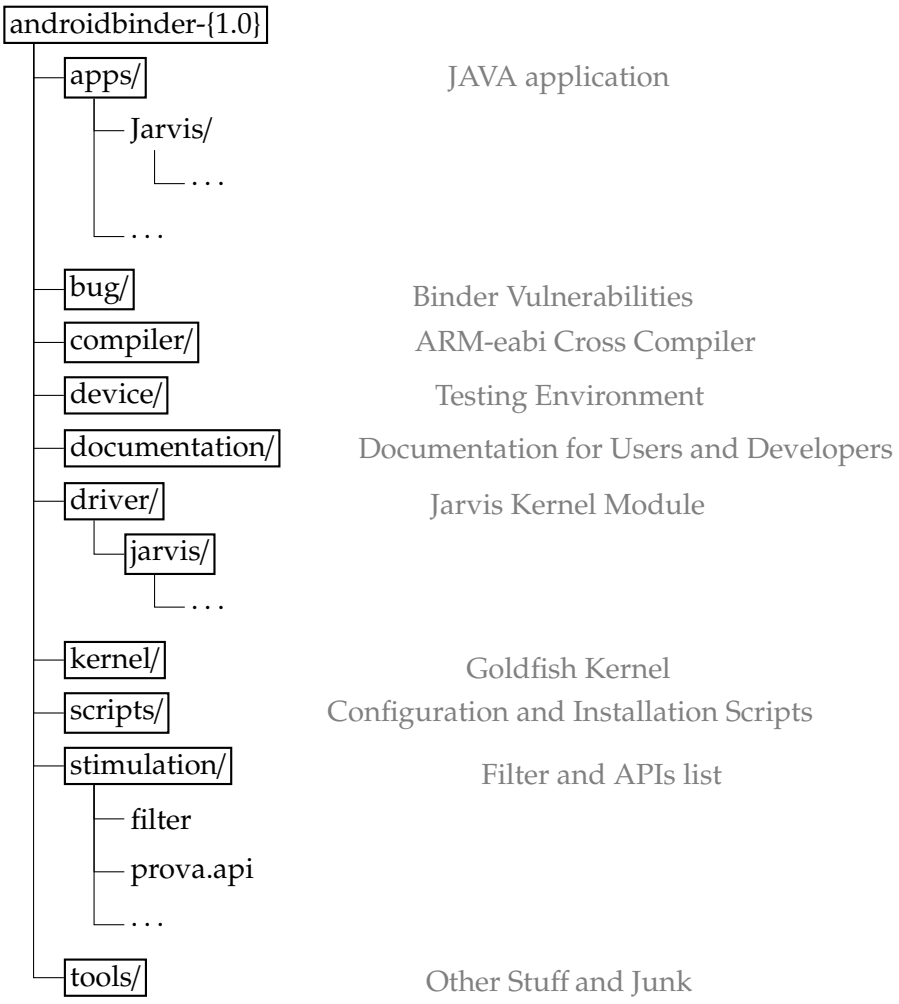

Figure 6.1: Project Repository

The Android application has been designed for Android 4.4.2 (version 19 of SDK). It requires a minimum API level of 8, corresponding to Android 2.2.

The Linux Kernel is taken by *Goldfish* project: "a family of similar virtual hardware platforms that mostly differ in the virtual CPU they support. Although it started as an ARM-specific platform, has now been ported to x86 and MIPS virtual CPUs" [12].

The testing environment was a device emulator (WVGA800).

6.2 Kernel Module

The main component we developed is a KLM [1], written in C, whose objective consists in sniffing system calls in an efficient and transparent way. The module constitutes the core of Jarvis. It was the most difficult to implement and required a good knowledge of the low-level features of Linux kernel and Android Binder.

6.2.1 General Overview

For each system call to track, the driver declares:

- A **wrapper**, to embed the system call and perform logging operations;
- A **filter**, to decide whether to log or to ignore a system call;
- A **parser**, to construe the signature[2] of the function and, in some case (i.e. the `ioctl` on binder device), to recover the data passing through the kernel.

In the Figure 6.2 we can appreciate the graphical description of driver workflow. Every intercepted system call pass through the filter and can be either accepted or discarded.

In the first case, the driver creates a log entry with some general information - such as name, process and thread id - and other data related to that particular system call, then inserts it into a log list that is periodically flushes into a file.

In the second case, the driver returns control immediately to the "right" method. The functioning is not very different of other System Calls Monitor excepts for two key features:

1. The filtering mechanism;
2. The quality of data, because the driver is able to go through the signature and to capture also the messages exchanged by the application through the Binder.

6.2.2 System Call Interception

The **System Call Table** is an array in which Linux kernel stores the address of the system calls. Hence, exchanging the address means basically intercepting and deviating the system call invocation. In order to do this, the

[1] Kernel Loadable Module
[2] Parameters and return value.

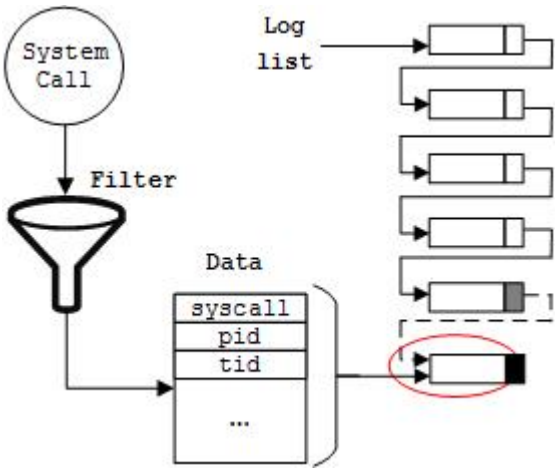

Figure 6.2: Driver Workflow: General Overview

driver substitutes the value of the location corresponding to tracked system call with the address of another function that we call generically *wrapper*.

The driver recovers the address of the table using a "magic" function. In the Figure 6.3 there is a summary of the interception process:

1. Before the loading of the driver, the system call table contains the "right" address. The driver stores wrappers addresses in an array of function pointers;

2. During `load()` function takes place the exchange of the addresses : the "right" one is put in a temporary buffer while the System Call Table is filled with the function pointer to the corresponding wrapper[3];

3. After the loading, the System Call Table points to the wrapper and the address of the system call is stored internally by the driver;

4. Whenever an application invokes a system call, the kernel can launch the wrapper, which provides to call the right system call and then return to its caller.

The interception is completely transparent to the user because logging functions recalls the original system call. Moreover, it allows to performs some operations both before and after its execution. In other words, this mechanism permits us to log both the parameters and the return value (and all related data).

The substitution of addresses - as well as the creation of input and output file in `/proc` file system - happens whenever the module is inserted into the kernel.

[3]This operation is performed for each system call that has a re-definition inside the driver.

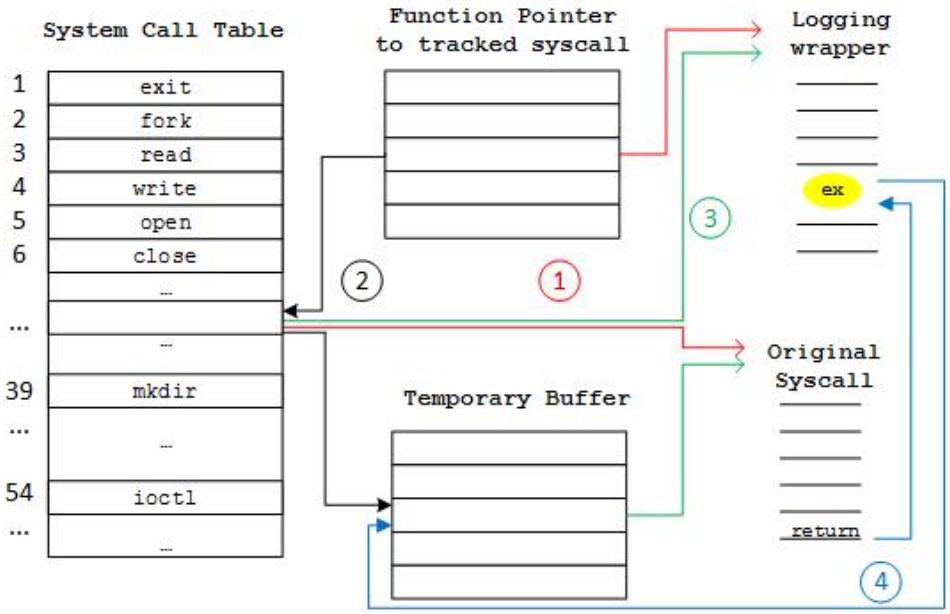

Figure 6.3: Interception of system calls

6.2.3 List of Tracked System Calls

The KLM is designed to be portable and expandable. A template (`logtemplate.c`) is available int the repository. It can be used to speed-up the process of adding new logging module. At now the tool deals with the following system calls:

- File System;
  - `open`;
  - `access`;
  - `mkdir`;
- Process Management;
  - `fork`;
  - `clone`;
  - `exec`;
- Network Communication;
  - `socket`;
  - `connect`;
  - `bind`;

- Special Android Device (Binder IPC);
  - `ioctl`;

6.2.4 I/O Control on Binder Device

The first goal of our project is to understand and clarify all the questions related to data binder transaction, particularly:

- The low-level protocol;
- The way in which process serializes and de-serializes objects;
- The link between `ioctl` calls on binder device and higher-level API.

For this reason, the tracking mechanism of `ioctl` system call is much more elaborated than the others. It explores in depth binder data transaction in order to log, among the system call's signature:

- The code of high-level method;
- The binder used to perform the transaction;
- The data exchanged through the kernel[4].

6.2.5 Filtering

In addition to the particular treatment of `ioctl`, the filter mechanism is the functionality that makes Jarvis special.

The filter allows cleaning the log from system calls not directly related with the APIs, like those of Graphical User Interface and of system log. The driver starts to log when the filter file is pushed in.

The mechanism is very flexible: there is a general parser that takes care of deciding whether to log a determined system call or not, and for each one there is a user-defined parser.

Actually the filter syntax is firewall-based. In particular, it replicates the rules of first generation firewalls based on packet filtering:

1. The firewall (filter) keeps no state. The filtering decision is made separately for every packet (system call), and does not take into account any earlier decisions made on related packets (system calls).
2. The filtering decision is based only of the five basic fields such as Source and Destination IP addresses, Protocol, and Source and Destination Port numbers (for protocols that have port numbers) [21]. In

[4]Data are serialized (i.e. flattened), so they need an interpreter.

our case the filter is based firstly on the system call name and on some general parameters like the process id and the thread id, then on a specific filter whose syntax is defined in the filter file.

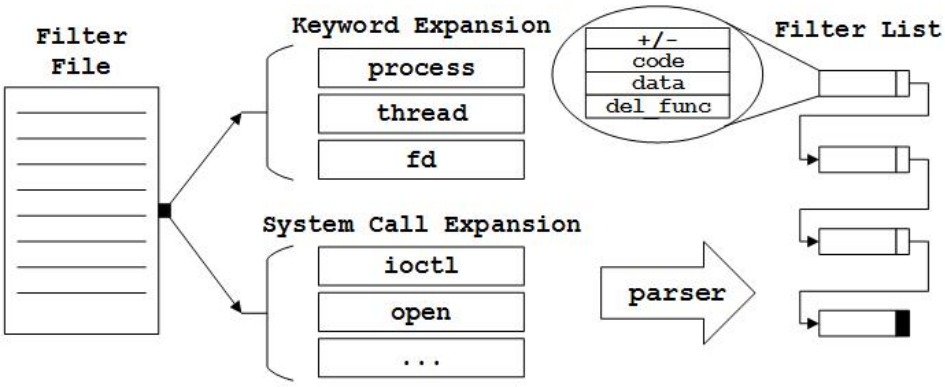

Figure 6.4: Filter Creation Process

The Kernel module creates a list of entries during installation phase by parsing the filter file. An entry has four data field containing:

- A flag (+ o -), which indicates the type of the entry: if the system call matches with the entry content, then it is included (+) or discarded (-) based on the content of this field;
- A code that contains the ID of the keyword or the system call;
- A pointer to a memory area containing data used by the filter;
- A function pointer to a method used to deallocate the previous buffer.

The actions that the filter can take if there is a match are:

- **Pass**: let the system call through;
- **Drop**: do not log the system call.

6.2.6 Logging

The logging mechanism is very simple: for each tracked system call the driver inserts the data in a queue periodically flushed in the file system (i.e. when a memory page is full).

To store data we use `seqfile` utility. It provides a safer interface to the `/proc` [5] filesystem than other libraries because it protects against overflow

[5]It's a virtual file-system that doesn't contain 'real' files but runtime system information (e.g. system memory, devices mounted, hardware configuration, etc). For this reason it's can be regarded as a control and information center for the kernel. The reading/writing operations are very fast.

of the output buffer and easily handles `procfs` files larger than one memory page.

Moreover, it supplies methods for traversing a list of kernel items, iterating on that list and also output facilities that are less error-prone than the previous interfaces.

The log is stored in kernel memory for three precise reason:

1. the memory size is incredibly greater than that available to an application;
2. storing data into user memory should require a new transition (Kernel $\Longrightarrow$ User), that is new system calls with a greater overhead and the problem of recursion;
3. performance (efficiency and speed).

6.3 Android Applications

The Java part of the tool is composed by a small Android application that deal with some fundamental tasks regarding data interpretation, APIs invocation and mapping. It is the other pillar on which Jarvis rests. Herein lies the difference between this tool and others System Calls Monitors. The Figure 7.1 shows the general structure. It exploits the same libraries to provide:

- High-to-Low mapping;
- Low-to-High rebuilding;

6.3.1 High-Level Schema

The application presents a very simple `MainActivity` used for:

- Starting and Stopping the driver (`DriverHandler`);
- Invoking a a collection of APIs at runtime (`Caller`, `Tester`, ...);
- Building an high quality log, organized by APIs and System Calls (`Logger` and derived modules);
- Inferring the high-level behavior given a log and the mapping (`Rebuilder`, `MappedAPI`)
- Reconstructing complex objects from the raw stream captured by the driver (`IoctlBinder`).

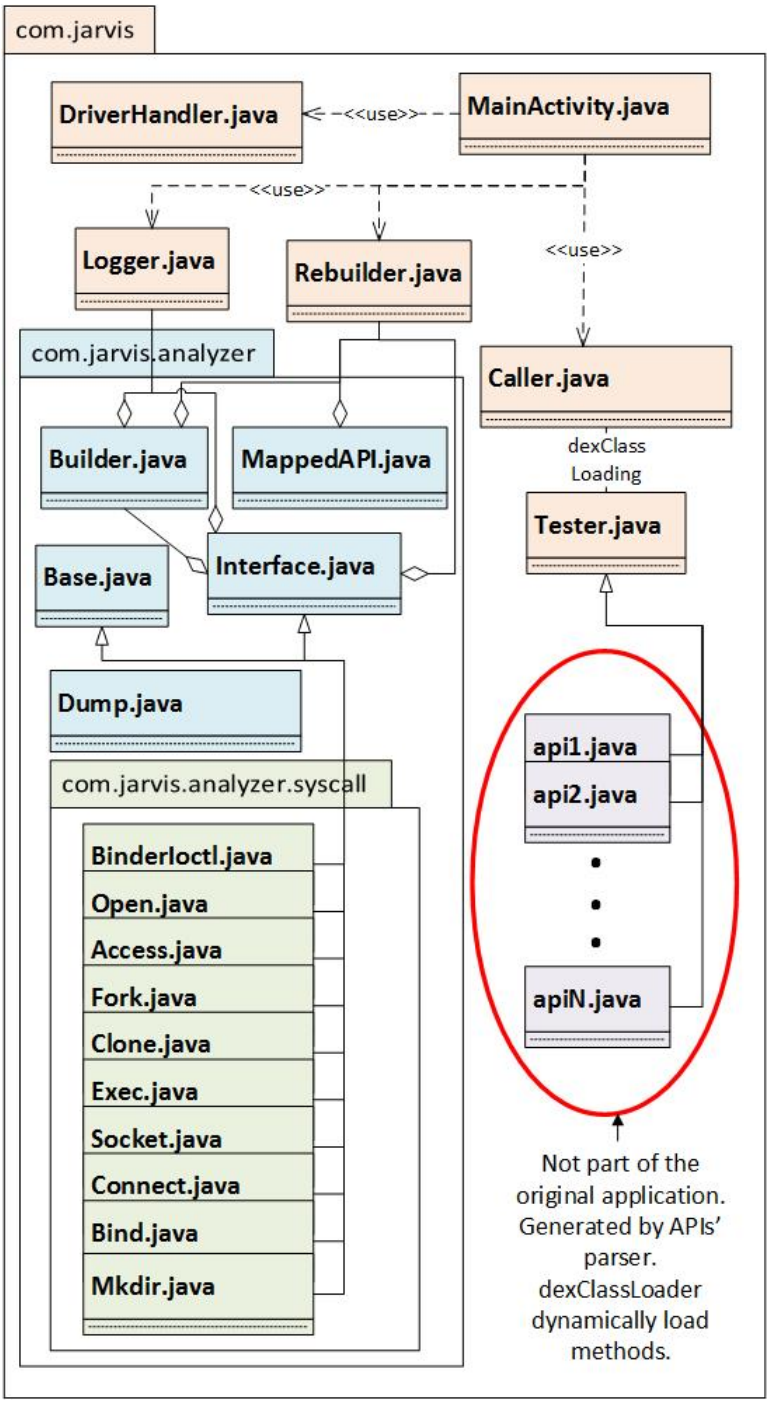

Figure 6.5: Main Components of Android Application

6.3.2 Data Interpretation

The Kernel module produce a bunch of raw data. At low-level we cannot re-built byte streams that are significant only at high-level. Hence, we need to exploit all the power of Java language and the facilities of Dalvik Virtual Machine. In particular, we need to deserialize data passing through Binder IPC and the way to do this is to recover API-specific Parcels by means of Java reflection.

Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the JVM (DVM maintains this feature). From the documentation ([16]): "This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language. With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible".

We can exploit Java reflection because Binder Protocol set the callee interface name as first parameter of a Binder Data Transaction. Starting from it we can recover the class, the function, the arguments and return values. If some data type is complex, we load them using Parcel.

Logging Capabilities

For application developers, the AIDL is a useful tool that avoids writing all the code needed to decompose objects into primitive data types that the operating system can understand, and "marshal" the object across that boundary. Nevertheless, it's not mandatory. Hence a programmer can freely decide to implement its own IPC and interact directly with the Binder driver using some customized native code in C++. In this case, it might be possible that the content of the data buffer (i.e the content of the Parcel) is different from the standard format. In this case, our application cannot parse raw data logged by the driver.

Another open issue is obfuscated code. **Obfuscation** in Java limits the use of the Reflection application programming interface and makes really hard or at worst impossible to load classes, methods and Parcels.

In the Table 6.1 we summarize the Binder calls we can log.

6.3.3 Mapping

Dalvik Virtual Machine allows to perform custom class loading. Instead loading Dalvik executable `dex` files from the default location, an application can take them from alternative locations such as internal storage or over the network. More in detail, Android `dexClassLoader` can load classes from `.jar` and `.apk` files containing a `classes.dex` entry. This can be used to execute code not installed as part of an application.

Binder Call Type	Logging Capability
Standard Binder calls to/from standard Android libraries or Framework Services	✓
Binder calls managed by automatically generated code (AIDL)	✓
Customized Binder calls (Stand-alone Protocol)	X
Obfuscated Binder calls	X

Table 6.1: Logging Capabilities of Jarvis

Jarvis contains a Python scripts that dynamically creates such **j**ava **ar**chive parsing a file containing the list of APIs and push it into the system. Once the `.jar` file is loaded, the application can recall the methods using an *ad-hoc* interface.

6.3.4 Re-Building

The application can recognize APIs for which it has the mapping (i.e. the list of corresponding system calls with parameters and return values).

Actually Jarvis utilizes a class to store the sequences of system calls instead of a SQLlite database for ease of implementation, execution speed and reduction of the size of available mapping.

The Re-Building process tries to match the logged system calls with those contained in the mapping, emitting the API only if all the methods in the sequence coincide in ordered succession.

6.4 Scripts and Utilities

The configuration scripts comprehends:

- a builder, which takes care of setup and compilation of Kernel module and of building Android application;
- an installer, which is responsible of:
  - Creation of the emulated device;
  - System bootstrap with goldfish kernel;
  - Making of a special directory where to put the driver module and the filter file;
  - Insertion of the module into the kernel;
  - Parsing of APIs' list file and realization of Java module for their invocation;

- Insertion of the dex-complied `.jar` into the device emulator.

The choice of using Python is given by the greater power of this scripting language with respect to Bash.

Chapter 7

Implementation

In this Chapter we go into implementation details of the tool, focusing on the more delicate aspects such as log list, filter management and data de-serialization by means of "dexLoaded" Parcels. The analysis of each part of Jarvis will be done with a *top-down* approach, from the "main" function to single modules and libraries.

7.1 Kernel Module

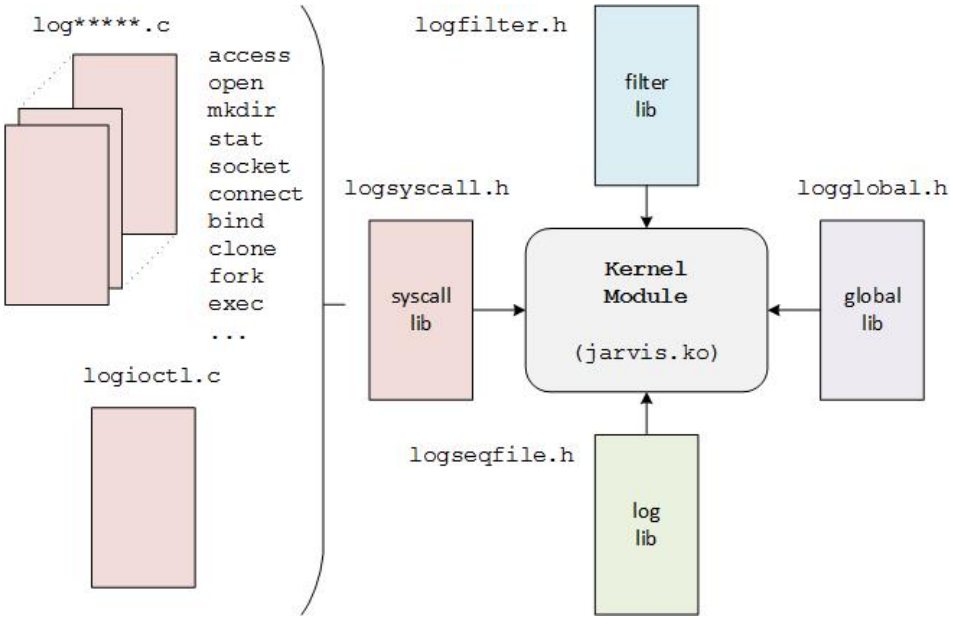

Figure 7.1: Main Components of Kernel Module

The driver provides some utilities to filter the data by the kernel and to

print them in a file. It uses the `procfs` library to exchange data at user level and a `seq_file` to write the data on a file that can be read at user space, and the normal write file system operation to read informations concerning filtering.

7.1.1 Load and Unload Function

The `load` method of the driver creates in the directory `/proc/` the `jarvisLog` file where it prints the data and the `jarvisFilter` where it recovers filter directives. It takes care of the substitution of system calls addresses, keeping the old ones in an array in order to re-call them inside the respective wrapper:

```
const char* read_file_name = "jarvisLog";
const char* write_file_name = "jarvisFilter";
int num_syscalls;
void* old_syscalls[];
void* new_syscalls[];
unsigned long *sys_call_table;
```

```
static int __init load(void) {
  struct proc_dir_entry *read_entry, *write_entry;
  num_syscalls = length(tracked_syscalls);
  read_entry = create_proc_entry(read_file_name, ...);
  ...
  write_entry = create_proc_entry(write_file_name, ...);
  ...
  sys_call_table = get_sys_call_table_addr();
  for (i = 0; i < num_syscalls; i++) {
    j = tracked_syscalls[i];
    old_syscalls[i] = (void*) sys_call_table[j];
    sys_call_table[j] = (unsigned long) new_syscalls[i];
  }
  return 0;
}
```

The `unload` method closes the `proc` file previously opened and restores the "right" addresses on the System Call Table:

```
static void __exit unload(void) {
  remove_proc_entry(read_file_name, ...);
  remove_proc_entry(write_file_name, ...);
  for(i = 0; i < num_syscalls ; i++) {
    j = tracked_syscalls[i];
    sys_call_table[j] = (unsigned long) old_syscalls[i];
  }
}
```

7.1.2 Global and System Call libraries

The driver is self-contained and make a strong use of global variables and C preprocessor macros. In particular, the drivers declares:

- the filter and the log list:

```
typedef struct safe_list {
  struct list_head* list;
  struct mutex* mutex;
} safe_list;
extern safe_list syslog;
extern safe_list sysfilter;
```

- the number of system calls and of general keywords:

```
extern       int   num_syscalls;
extern const int   num_keywords;
```

- the arrays of addresses of the system calls and of the wrappers:

```
extern void*       new_syscalls[];
extern void*       old_syscalls[];
```

- the arrays of function pointers to filter and parser methods:

```
extern void (*keywords_parser[])(char*,int);
extern void (*syscalls_parser[])(char*,int);
extern bool (*syscalls_filter[])(void*,void*);
```

The `syscall` library contains the data structures containing that define log information for each system call

7.1.3 SeqFile library

`SeqFile` is the output library of the Kernel module. In order to speed-up logging process the driver manages a queue of entries, inserting in tail one element for each logged system call and flushing it periodically starting form the front to keep the order. The list is a shared resources so it is protected by a mutex semaphore.

```
LIST_HEAD(log_list);
DEFINE_MUTEX(log_mutex);
```

The entry is designed to be flexible and to occupy less bytes as possible:

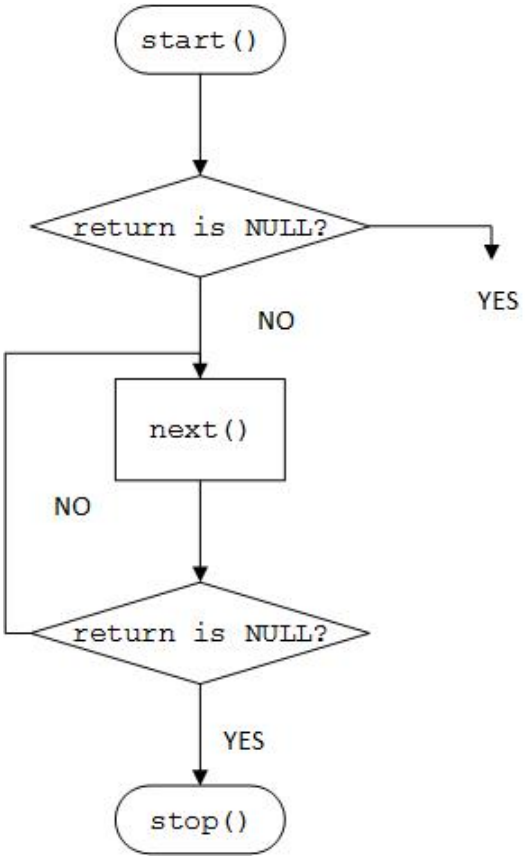

Figure 7.2: SeqFile Algorithm

```
typedef struct printable {
  struct list_head list;
  void *data;
  void (*print)(struct seq_file*, void*);
  void (*del)(void*);
} to_print;
```

The `to_print` data structure stores the memory buffer in which logged data are contained and two function pointers:

- `print`, which specifies the way of printing data;
- `del`, which defines the way of deleting data after being flushed to the file.

The methods above are specific for each system call, while the following one serves to allocate and initialize `to_print` data structures.

```
void prtelem(void* data, void (*print)(struct seq_file*,
    void*), void (*del)(void*)) {
  to_print *p;
  ALLOC(p,to_print,sizeof(*p),del(data);return);
  p->data = data;
  p->print = print;
  p->del = del;
  mutex_lock(syslog.mutex);
  list_add_tail(&(p->list), syslog.list);
  mutex_unlock(syslog.mutex);
}
```

Whenever a user tries to open the `jarvisLog` file, the Kernel executes the algorithm shown in Figure 7.2, where `start`, `next` and `stop` are implemented to handle the log list.

7.1.4 Filter library

The filtering unit contains the procedures to parse the related file and to support this activity at runtime. The filter file has a very simple syntax:

```
@name{ ... }
```

The `@` character indicates the starting of a new section, the `name` can be either a system call or a particular keyword such as `process`, `thread`, `fd` and the brackets `{}` wrap the content of the filter.

Whenever a writing operation is invoked on the filter file, the driver cleans the previous filter list and creates a new one by means of a simple method that scans the file looking for the special character and then invokes a function that recognizes the keyword and calls the name-specific parser.

```
void parser(char* file, int len) {
  int i = 0, ret;
  while(i < len) {
    ret = findchar(file + i,len - i,'@'));
    i += ret;
    ...
    ret = check_and_call(file + i,len - i)) < 0)
    i += ret;
  }
}
```

Indeed, each keyword/system call defines its own parser, which takes care of creating a set of entries of the following type:

```
typedef enum filtype { PLUS, MINUS } filtype;
struct filter {
  struct list_head list;
  filtype t; // PLUS = include, MINUS = discard
  uint32_t keyword;
  void* data;
  void (*del)(void*);
};
```

and inserts it into the filter list. The library provides this method to automate insertion process:

```
void fltrelem (filtype t, uint32_t k, void* data, void (*
    del)(void*)) {
  struct filter* elem;
  ...
  elem->t = t;
  elem->keyword = k;
  elem->data = data;
  elem->del = del;
  ...
  list_add_tail(&(elem->list), sysfilter.list);
}
```

The choice of delegating to the specific parser the creation of the filter list permits the definition of different syntaxes, calibrating the filter mechanism based on keyword-specific data.

When the driver log a system call, it scans the list looking for the corresponding `keyword` (the system call name or a general one). If there is a match, then the specific filter is recovered and it compares in such a way the filter directives (parsed from a file) with the `data` pointed by the respective field.

```
LIST_HEAD(filter_list);
DEFINE_MUTEX(filter_mutex);
```

The library contains also some utility to handle basic filtering operation.

7.2 I/O Control Log

A brief analysis of the module that handles `ioctl` allows us to describe some general features of the system calls logging modules and to explain how to completely wrap a system call using simple assembly program[1].

[1]We have used this technique also for process-related system call such as `fork()`, `clone()` and `execve()`, in which the forking required an *ex-post* evaluation.

7.2.1 Common Functions

As all other similar library, `ioctl.c` contains the declaration and the definition of five fundamental functions:

- `void ioctl_parser(char* str, int len);`
- `bool ioctl_filter(void* data, void* own);`

  These two function are addressed by the function pointers arrays declared in the global library.

- `void print_ioctl(struct seq_file* s, void* ptr);`

  The address of this function is stored in a `to_print` data structure whenever a `ioctl` system call passes the filter.

- `void del_filter(void* data);`
- `void del_ioctl(void* ptr);`

  These functions takes care of de-allocation of the structures related to respectively the filter and the log data.

7.2.2 Assembly Routine

The `ioctl` is a system call that requires a complete log for our purposes, in the sense that the driver must wait the end of the procedure in order to capture some important information like return value or, in this case, data sent back to client in two way transaction [2].

In order to solve this kind of problem we had to create a small ARM Assembly routine to:

- identify an `ioctl` call with "`/dev/binder`" as target;
- call the log for the write transaction, which could be called before the `ioctl`;
- invoke the "real" system call, saving the return value;
- call the log for the read transaction, which could be done only at the end of `ioctl`, when the data were available to the user;
- restore the return value of the "real" system call;
- pass the control to the system call handler.

[2] Data are available only at the end of the system call.

```
.global new_ioctl;
.set BINDER, 3;
.align 8;

new_ioctl:
  push {r4-r7,lr};
  mov r4,r0;
  cmp r4,#BINDER;
  bne old;

  push {r0-r3};
  bl pre_ioctl;
  mov r6, r0;
  pop {r0-r3};

old: ldr r7, =old_ioctl;
  ldr r7, [r7];
  ldr r7, [r7];
  blx r7;
  cmp r4,#BINDER;
  bne end;

  mov r5, r0;
  mov r0, r6;
  bl post_ioctl;
  mov r0, r5;

end: pop {r4-r7,pc};
```

Once created this routine, it was easy, knowing the articulate protocol of the binder `ioctl`, to log all the key data.

7.2.3 Log data

Each tracked system call declares a container for the logged data. In this case:

```
struct transaction {
  uint32_t        syscall;
  uint32_t        total;
  pid_t           pid;
  pid_t           tid;
  uint32_t        function;
  int             arg_size;
  int             reply_size;
  unsigned char*  arg_buf;
  unsigned char*  reply_buf;
  uint32_t        checksum;
};
```

Some fields like the system call code (`syscall`), the process and thread id (`pid`, `tid`), the length of data (`total`) and the checksum (`checksum`) are common while some others like the function code (`function`), the request and reply buffer (`arg_buf` and `reply_buf`) and the respective size (`arg_size` and `reply_size`).

7.2.4 Filter Syntax

Each system call can be filter in a specific way by means of the couple of functions `## syscallname ## _parser` and `## syscallname ## _filter`.

The former is used to parse the content of filter file related to the specific keyword, the latter is invoked runtime during filtering process to compare the logged data with those obtained by parsing the filter file.

According to Binder Communication Protocol, the first argument of each data transaction is the interface token or rather the package name, hence we retained a good choice organizing the filter file as a list of pairs <+/- | package_name>:

```
[+/-]interface_path1
[+/-]interface_path2
[+/-]
[+/-]interface_pathN
```

The `interface_path` specifies a package or a group of packages that has to be included (excluded) by the log operations.

For example, the row:

```
-com.android
```

excludes all the interfaces whose name starts with the string `com.android`, viceversa, the row:

```
+
```

includes all the interfaces by default.

The parser creates for each pair an entry in the filter list initialized as following:

Field	Data
type	+/-
keyword	ioctl
data	interface_path
del	del_filter

[3]

We use the same criteria of firewall engine, it means that the filter operations stop at first row that matches the name of the interface, so the order is fundamental.

7.3 Java Applications

At the top-level the `MainActivity` manages two fundamental operations:

- **High-To-Low**

[3]The `ioctl` keyword is a code associated to the name of the system call.

```
while(apiTest.hasRemaining()) {
  driver.start();
  apiTest.execute(this);
  driver.stop();
  ...
  log.doLog();
}
```

- **Low-To-High**

```
Intent intent = new Intent(Intent.ACTION_MAIN, null);
ComponentName cn = new ComponentName("com.example.
    toysample","com.example.toysample.MainActivity");
intent.setComponent(cn);
...
driver.start();
startActivity(intent);
... // On received intent
Intent i = getIntent();
int id = i.getIntExtra("ID", 0);
...
driver.stop();
reb = new Rebuilder(PROCDIR + LOG);
reb.doReb();
```

7.3.1 Driver Handler

Logging process does not begin till the filter file is empty. Therefore, the driver needs to be managed by an handler, which takes care of starting and stopping it.

```
public void start() {
  ...
  String cmd = "cat␣" + filterFilePath + "␣>␣" +
     filterDriverPath;
  String[] args = new String[]{shell,"-c",cmd};
  Process p = Runtime.getRuntime().exec(args);
  ...
}
public void stop() {
  ...
  String cmd = "echo␣\"␣\"␣>␣" + filterDriverPath;
  String[] args = new String[]{shell,"-c",cmd};
  Process p = Runtime.getRuntime().exec(args);
  ...
}
```

7.3.2 Logger

The driver save the high-to-low mapping in its folder "`/data/jarvis`" using different file for each API. The logger class takes care of withdrawing raw data from the driver log file and print it in the appropriate output file.

The first four bytes of the log are used as discriminant of the type of the system call, then the buffer is passed to the appropriate class that handles the printing. We use the *Abstract Factory* method in order to generalize the creation of specific system call printer. Going into detail, all system call printers implement a common interface (`Interface`), which is returned by a factory (`Builder`) that uses system call code to decide which is the class to build.

```
private FileInputStream is;
private FileOutputStream os;
private String className;
public void doLog() {
  if (is == null || os == null) {...}
  byte[] buf = new byte[Base.SIZEOFINT];
  ...
  while(is.read(buf)>0) {
    int s = ByteBuffer.wrap(buf).....getInt();
    SyscallIndex magic = SyscallIndex.fromValue(s);
    Builder factory = new Builder(magic,is,os);
    Interface syscall = factory.GetInstance();
    ...
    syscall.print();
}
```

7.3.3 Rebuilder

The same utilities are used in the reverse process, which is divide in two phases:

- checking the type of system call;
- comparison of the contents.

Passing both control allows handler to push forward into the sequence till the API is matched. The first comparison checks only the type of system call, in order to immediately discard different system calls. The second comparison happens using, for example, the interface name in case of Binder call or the file path in case of `open`. As a general rule, the logged system call and the next element of the API's sequence must match for each data not runtime-dependent.

```
for (MappedAPI api: apis) {
  if (index == api.getSysIndex()) {
    res = syscall.compare(api.getData());
    if (res) {
      api.incStatus();
      if (api.match())
        recognized.add(api.getName());
    }
  }
  ...
}
```

7.3.4 Caller

The most elaborated part of the application concerns APIs invocation. In order to do high-to-low mapping, Jarvis loads runtime a "stimulation" `.jar` file, in which all the APIs are embedded, by means of `DexClassLoader`: the Android-specific class loader that can recover Java classes from `.jar` or `.apk` file with a `classes.dex` entry.

The external archive must to be re-compiled using `dx` tool.

During the setup phase the driver loads `.jar` file and stores in a queue the set of APIs to map.

```
public Caller(String dexFilePath, String dexDir,
    ClassLoader loader) {
  ...
  names = new ArrayList<String>();
  apis = new ArrayList<Class<?>>();
  Enumeration<String> en;
  ...
  dexFile = DexFile.loadDex(dexFilePath, dexDir + "/
      dexCache", 0);
  dexLoader = new DexClassLoader(dexFilePath, .. , loader);
  for (en = dexFile.entries(); en.hasMoreElements(); ) {
    String className = en.nextElement();
    Class<?> tmp = Class.forName(className,true,dexLoader);
    apis.add(tmp);
    names.add(className);
    ...
  }
}
```

During the execution phase, the method loaded using Reflection is invoked:

```
public void execute(Activity activity) {
  inExecution = names.remove(FIRST);
  Class<?> tmp = apis.remove(FIRST);
  Class<?>[] params = new Class[] {Activity.class};
  String method = "call";
  Method m = tmp.getDeclaredMethod(method,params);
  m.invoke(tmp.newInstance(),activity);
}
```

7.3.5 System Call Log and Object Deserialization

The code of Java classes that manages high-level data interpretation is very simple: it takes the data from an input byte stream and print it in a more elaborated way using an output stream. The most complex part concerns the parsing of data buffer from Binder in `BinderIoctl` class. The following function handles object type and data loading:

```
private Object loadObject(Class<?> type, Parcel parcel) {
  Object myObj;
  if(type.equals(Byte.TYPE) || type.equals(Byte.class))
    myObj = parcel.readByte(); // Basic Types
  ...
  else if(type.isArray()) { // Array
    if(type.equals(String[].class))
      myObj = parcel.createStringArray();
    ...
  }
  ...
  else { // Object
    Field field = type.getDeclaredField("CREATOR");
    field.setAccessible(true);
    Parcelable.Creator CREATOR = (Parcelable.Creator) field
        .get(null);
    myObj = CREATOR.createFromParcel(parcel);
    ...
  }
  return myObj;
}
```

7.4 Stimulation and Mapping

Building and installation process are tedious, so we have automated it using two bash scripts. The key component is a Python program who takes care of the creation of APIs' wrappers, their compilation and loading into the device. The creation of wrappers starts from a template:

```
template =
""" package com.jarvis;
import android.app.Activity;
{0}
public class {1}Tester {{
  public void call(Activity activity) throws Exception {{
    {2}
  }}
}}"""
```

The script parses the file in which APIs are listed and replaces `{0}`, `{1}`, `{2}` with, respectively:

- the package needed to invoke API;
- the name of the API, which is concatenated with "`Tester`" to form the name of the testing class;
- the code.

These information have to reside in the `.api` files, inside the `stimulation` folder. We pass as argument a reference to an activity in order to make available an application context, which is fundamental, for instance, to look for system services.

```
dir = "stimulation"
for f in os.listdir(dir):
  if f.endswith(".api"):
    tmp = open(os.path.join(dir,f))
    api = tmp.read().split("\n")
    tmp.close()
    imports = "\n".join([i for i in api if i.startswith("
       import ")])
    code = "\n".join([i for i in api if not i.startswith("
       import ")])
    javacode = template.format(imports,f.replace(".api","")
       ,code)
    javafile = open(os.path.join("src/com/jarvis",f.replace
       (".api","Tester.java")),"w")
    javafile.write(javacode)
    javafile.close()
```

The compilation is split in two phase: the first using classical `javac` to creates bytecode files, the second to adapt the archive for Android Class Loader:

```
os.mkdir("obj")
cmd = "javac -verbose  -d obj -classpath $ANDROID_SDK/
    platforms/android-19/android.jar:obj -sourcepath src/
    src/com/jarvis/*.java"
subprocess.call(cmd,shell=True);
cmd = "$ANDROID_SDK/build-tools/19.0.0/dx --dex --output=
    stimapp.jar obj/"
subprocess.call(cmd,shell=True);
subprocess.call("rm -r src/ obj/",shell=True);
```

Chapter 8

User Guide

In this Chapter we describe the right way to use Jarvis: compilation and setup process, configuration and syntax of filter file, building and installation. Jarvis is a newborn tool and has ample room of improvement. Some design choices permits the implementation of new facilities quickly and easily: we explain the way to implement new module using the library of the tool and to define a different syntax for the filter.

8.1 Setup

Jarvis is available at the git repository [20] and can be downloaded upon request to the authors using the `clone` command of git tool. The most important folders are:

- `apps`, which contains the Android application;
- `driver`, which holds the Kernel module;
- `scripts`, which contains the configuration scripts.

Although the tool was designed to run on Android KitKat and "Goldfish", it can be ported on different Kernel and also on different emulators or devices.

8.1.1 Building phase

The compilation of kernel module and Android application is completely automatic. The user must only set the environment variable related to kernel source code location in the building script, which must be launched from the root of the project folder.

```
tom@Mappamondo:~$ cd $ROOT
tom@Mappamondo:~/Projects/androidbinder$ ./build
```

8.1.2 Installation phase

The installation of Jarvis into the emulator is completely automatic as well. The script set the execution environment creating the device with the working directory and pushing in the kernel module, the filter file and the java archive with with APIs to stimulate (used only in case of high-to-low mapping).

```
tom@Mappamondo:~/Projects/androidbinder$ ./install
```

8.1.3 Usage phase

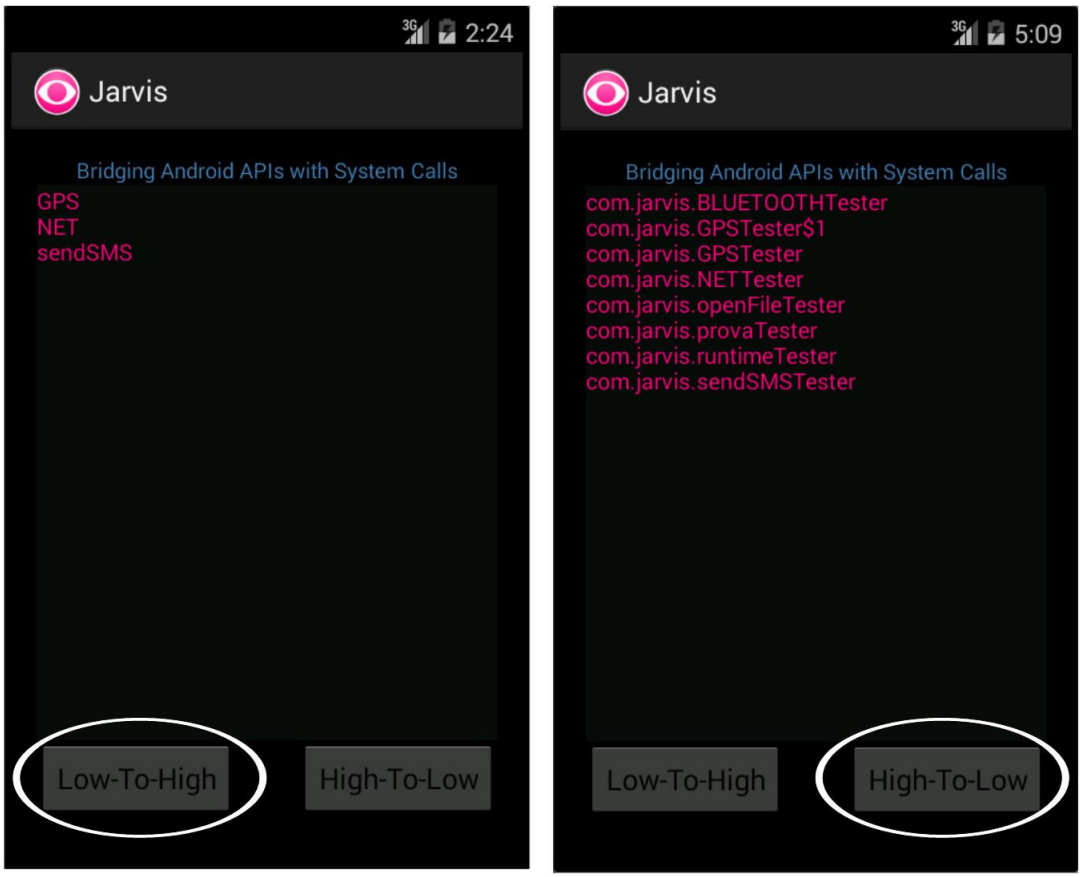

Figure 8.1: Main Activity of Jarvis Application

The graphical interface is minimal, with two buttons to launch the mapping in both directions (high-to-low and low-to-high) and a view to print the results:

- the list of tracked APIs, in case of high-to-low mapping (Figure 8.1);
- the list of recognized APIs, in case of low-to-high mapping.

8.2 Adding System Call Logging Library

To implement a log of a system call, the developer has to write a file in which to implement the logging function. The convention for the name is: `log ## syscall_name ## .c`. Each new file must be added to the Makefile of the module.

8.2.1 Header file

The header file must include the main libraries:

- `logglobal.h`
- `logseqfile.h`
- `logfilter.h`
- `logsyscall.h`

Particularly, it should declare a data structure in the last library to define the format of the data to print. From this point forward we refer to this structure as **container**. Some fields are mandatory:

```
struct syscall_name {
  ...
  uint32_t      syscall;
  uint32_t      pid;
  uint32_t      tid;
  ...
};
```

where:

- **syscall** must be initialized with `YOUR_SYSCALL_NAME`;
- **pid** must be initialized with the identifier of the current process;
- **tid** must be initialized with the identifier of the current thread.

8.2.2 Source file

The source file should contain the following method:

- A function that log the data and insert them in the wrapper.

```
void print_ ## syscall_name(struct seq_file* s, void*
    ptr);
```

  This function should do the following key operations:

  – Allocation of the memory needed by the following data structure (defined in `logseqfile.h`):

```
typedef struct printable {
  struct list_head list;
  void *data;
  void (*print)(struct seq_file*, void*);
  void (*del)(void*);
} to_print;
```

    that must be initialized in the following way:

    * `data` = pointer to the wrapper (which must be dynamically allocated)
    * `print` = function pointer to the method that specifies the way to print your data in the file read by the user.
      The method must be composed by two arguments:
      1. The first argument is the file;
      2. The second is the pointer to the wrapper [1].

      and can utilize this subroutine:

```
void inline print(struct seq_file* s, void* elem
    , int len);
```

      that print in the `seq_file s` the buffer pointed by `elem` of length `len`.
    * `del` = function pointer to the method that specifies the way to delete the wrapper. If you do not allocate anything except the wrapper, this function should not do anything, so it's sufficient to define it as the following:

```
void del_ ## syscall_name (void* ptr) {return;}
```

[1] This pointer is generic (**`void*`**, so it must be cast before executing any print operation

– Insertion of the structure in the tail of log list. It is reachable, with the corresponding mutex by means of a global variable:

```
typedef struct safe_list {
  struct list_head* list;
  struct mutex* mutex;
} safe_list;
extern safe_list log;
```

- A function that parses the system-call related content of the filter file.

```
void syscall_name ## _parser(char* str, int len);
```

It takes care of:

– Scan each row/section of the specific part of the file;

– Create one of more entries of following type:

```
typedef enum filtype { PLUS, MINUS } filtype;
struct filter {
  struct list_head list;
  filtype t; // PLUS = include, MINUS = discard
  uint32_t keyword;
  void* data;
  void (*del)(void*);
};
```

to insert in the filter list. The data fields must be initialized in the following way:

* `t` = type of the filter element. It indicates the action to do if there is a match between its content and that of the logged data;
* `keyword` = the code of the special keyword or the system call;
* `data` = the content of the filter element: it could be a string, a number, a complex type;
* `del` = the function that deallocates the memory buffer pointed by `data`. If there is not complex data structure to free, it can be a simple **`return`** instruction.

```
void del_filter(void* data);
```

- A function that filter the data captured by the driver and decide whether to print them into the log or discard them:

```
bool syscall_name ## _filter(void* data, void* own);
```

This function should perform some comparison between the content of the filter and that of the logged data. There is an utility for string-type filter:

```
bool fltrcmp(const char* name, const char* filter);
```

that returns `true` if the substring of `name` is equal to `filter`, `false` value otherwise.

8.2.3 Modifications in global files

The user has to add few rows to a couple of files. Particularly:

1. `logglobal.h`
   - In the macro `FOREACH_SYSCALLS` an entry `SC(YOUR_SYSCALL_NAME)`. The convention is to use uppercase letters.
   - An entry `MY_SYSCALL(name, args_type_list);` with the following parameters:

     name: the lowercase name of the system call;

     args_type_list: the list of arguments type.
2. `logprocfs.c`
   - An entry at the end of array `tracked_syscalls` with the value `__NR_ ## your_syscall_name`. The `__NR_` macro is a kernel utility which holds the index of the syscall inside the table;
   - An entry at the end of array `new_syscalls` with the name of your log function, that is defined by the macro `MY_SYSCALL` as `new_ ## your_syscall_name`;
   - An entry at the end of array `syscalls_filter` with the name of your filter function, that is defined by the macro `MY_SYSCALL` as `your_syscall_name ## filter`;
   - An entry at the end of array `syscalsl_parser` with the name of your parser function, that is defined by the macro `MY_SYSCALL` as `your_syscall_name ## filter`.

8.2.4 Working to the log and filter lists

Our driver contains two global lists: one for the filter and one for the log. They are implemented using Linux library `list.h`, so each entry contains a field **`struct`** `list_head` and the code to handle insertion and extraction is very simple.

- Whenever we want to insert an element (a log structure) in the log list, we should use the following code:

```
prtelem(ptr, print_ ## syscall_name,del_ ##
    syscall_name);
```

- Whenever we want to insert an entry (a filter unit) in the filter list, we should invoke the following method

```
fltrelem(type,SYS__ ## syscall_name,data, del_filter);
```

We remember that the read file is used for the log (informations tracked) and the write file is used for the filter (firewall syntax).

8.3 Template

The driver folder contains a file (`logtemplate.c`) that is not compiled during build phase. It can be as starting point for a new module:

```
void print_ ## syscall_name(struct seq_file* s, void* ptr)
    {
  uint32_t checksum = 0;
  struct your_syscall_name *x = ptr;
  print(s,&(x->syscall),sizeof(uint32_t));
  checksum += x->syscall;
  print(s,&(x->total),sizeof(uint32_t));
  checksum += x->total;
  print(s,&(x->pid),sizeof(pid_t));
  checksum += x->pid;
  print(s,&(x->tid),sizeof(pid_t));
  checksum += x->tid;
  // ...
  x->checksum = checksum;
  print(s,&(x->checksum),sizeof(uint32_t));

}
```

```
void your_syscall_name_parser(char* str, int len) {
 /*
  * N.B. For each element
  * Parsing
  * Allocation and initialization of your filter data
      structure.
  */
  fltrelem(PLUS,SYS__## syscall_name,/*filter data
      structure*/, del_filter);
}
```

```
asmlinkage long new_## syscall_name(...) {
  your_syscall_name old_your_syscall_name = old_syscalls[
      SYS__ ## syscall_name];
  pid_t pid = getpid();
  pid_t  tid = gettid();
  int fd = NO_INFO;
  // ...
  struct global data = { pid, tid, fd };
  struct your_syscall_name *x;
  uint32_t total = 0;
  void* to_cmp;
 /*
  * Allocation end initialization of your filter data
      structure
  */
  to_cmp = /* Pointer to your filter data structure */
  if(!filter(to_cmp,data,SYS__## syscall_name)) goto end_##
       syscall_name;
  total += sizeof(uint32_t); // For the checksum
  x->syscall = syscall(SYS__your_syscall_name); //
      Translation of id
  INS_VALUE(pid,x->pid,total,your_syscall_name,%u);
  INS_VALUE(tid,x->tid,total,your_syscall_name,%u);
  x->total = total;
  prtelem(a,print_your_syscall_name,del_your_syscall_name);
  return res;

end_your_syscall_name:
  return old_your_syscall_name(...);
}
```

8.4 Java Module

Jarvis high-level framework support the development of new log library in a similar way. It makes the following assumptions:

- The expected log format for each system call contains firstly the identifier, then the length of the data structure, pid and tid;
- Checksum stands at the end of each log, and must be control using little endian standard.

If they are consistent the Java class can assume the following structure:

```
public class MySyscall extends Base implements Interface {
  private ... ; // Private Field
  public MySyscall(FileInputStream is, FileOutputStream os)
      throws IOException {
    super(SyscallIndex.MY_SYSCALL,is,os);
    ... // Initialize private fields from tmp buffer
  }
  public void print() throws IOException {
    String s;
    super.printHeader();
    s =  ... ; // Private data
    os.write(s.concat("\n").getBytes(MODE));
    ...
    super.printTail();
  }
  public boolean compare(Class<?> syscall) throws
     IOException {
    if (syscall == this.getClass()) {
      ...
    }
    return false;
  }
}
```

Where the `Interface` contains the two methods used during the execution either in High-to-Low direction (`print`) or in Low-To-High one (`compare`), and the `Base` provides all the common data fields and the procedures for reading/writing by a stream.

Part III

Mapping API - System Call

Chapter 9

Binding High and Low Level Behavior

In this Chapter we describe the results of some initial tests done to verify the correct working of our tool, the goodness of the approach and the effectiveness of the filter mechanism.

9.1 Objectives

The objective of this dissertation were:

- understanding the low-level protocol of Binder Framework
- implementing Jarvis, a tool constituing a first trial of bridge the semantic gap between Android APIs and system calls;
- providing a description and a user guide for our tool;
- demonstrating that the rightness of approach, that is we can map some APIs and use these mapping to rebuild high-level behavior from the system calls;
- measuring the efficiency of some capability of the tool as the filter.

We proceeded in the following way:

1. we have a tool that given a set of APIs produces a detailed list of system calls that can be considered a sort of "signature" of the API itself;
2. we create the mapping for a number of APIs (that span over different interesting/meaningful categories), and we show that even for the binder-related ones the system gives useful information;

ID[2]	API	Interface
1	`sendTextMessage`	android.telephony.SmsManager
2	`getLastKnownLocation`	android.location.LocationManager
3	`openConnection`	java.net.URL
4	`FileInputStream`	java.io.File
5	`exec`	java.lang.Runtime

Table 9.1: List of APIs and Interfaces invoked in the test

3. we show how the output system calls are disjointed enough so that it's possible to "go back", from the system calls to the APIs.

The novelty would be that this is the first work that tries to understand if it's possible to automatically reconstruct the high-level behavior of an Android application.

9.2 Experiment Planning

Jarvis is able to perform the following operations:

1. System Call logging;
2. High-level data interpretation and de-serialization of Binder messages;
3. Android APIs invocation and mapping;
4. Reconstruction of high-level behavior given a system calls log and a collection of APIs ⇔ System Calls mapping.

The experiment is divided in two phases:

1. Using Jarvis in high-to-low direction to map some APIs **??**. The choices were made based on two considerations:

   - Logic relationship between APIs and tracked system calls [1];
   - Chance to log Binder Calls (Table 6.1), that is maximum for system services that fully utilize Binder Framework.

   The stimulation phase consists in APIs invocation, system calls logging, filtering and storing of the signature in terms of:

[1] If our tool comprehends network-related system calls, then we choose some APIs that should use it reasonably.

- System Calls list;
- Arguments and return values;
- Data exchanged through the binder.

2. Using Jarvis in low-to-high direction, exploiting the previous mapping and a Toy Sample that calls sequentially a list of either mapped or not Android APIs, looking for false positive or false negative.

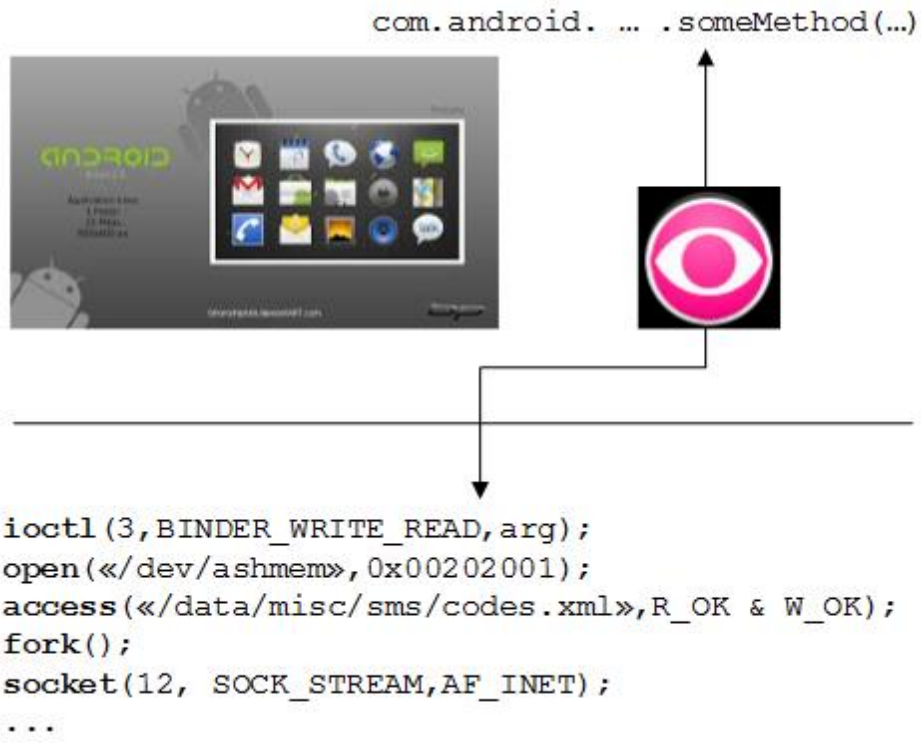

Figure 9.1: Jarvis Usage in Bidirectional Way

In rebuilding phase, we measure the performance of the filter to analyze the load on kernel depending on the number of APIs correctly recognized.

9.3 Testing Environment

All the tests have been made in an emulated environment because of:

- Difficulty to install a modified kernel in a real Android device (by default it is not possible to load external module);

- Objectives of the work, which are focused to understand if the approach and the ideas behind Jarvis are correct, does not specifically require a real environment.

9.3.1 Filter Settings

The filter is an important novelty of our tool. Despite the simplicity of the syntax, it permits discarding lots of useless - in terms of reconstruction of high-level behavior - system call, obtaining a substantial reduction of load on the Kernel Module (Table 9.8).

We have adopted only few basic filter, centered on the main "waste" of system calls, which regards in particular `open` and `ioctl`:

- `ioctl`:
  - Discarding all requests/responses to `android.gui.IGraphicBufferProducer`;
  - Discarding all requests/responses to `android.gui.DisplayEventConnection`;
  - Discarding all requests/responses to `android.ui.ISurfaceComposer`;
  - Discarding all requests/responses to `android.os.PowerManager`;
- `open`:
  - Discarding all files whose path starts with `/proc`, because it concerns mainly system and process log;
  - Discarding all files whose path starts with either `/vendor/lib` or `/system/lib`, indeed Android looks for system library (like libc.so and libstdc++.so) in both these folder, and recover them only from one;
- `access`:
  - Discarding all access to file `/.../gralloc.goldfish.so`, becuase it is related to GPU emulation.

9.4 Mapping API in System Call

The stimulation phase produced the mapping shown in Table 9.2, 9.3, 9.4, 9.5 and 9.6. We discovered that the number of system calls per APIs is quite large (some dozens), in we put in the tables only the pattern we use to re-build the high-level behavior.

System Calls	*Parameters*						
ioctl	**Interface** android.os. IService Manager	**Code** 1	**Argument** isms				
access	**Path** /data/misc/ sms/codes	**Pid** 564	**Tid** 881	**Uid** 1001	**Gid** 1001	**Euid** 1001	**Egid** 1001
ioctl	**Interface** android.os. IService Manager	**Code** 1	**Argument** phone				
access	**Path** /data/data/ com.android. providers. telephony/ databases/ mmssms.db -journal	**Pid** 564	**Tid** 564	**Uid** 1001	**Gid** 1001	**Euid** 1001	**Egid** 1001
open	**Path** /data/data/ com.android. providers. telephony/ databases/ mmssms.db -journal	**Pid** 564	**Tid** 564				

Table 9.2: sendSMS(...) API mapping

System Calls	*Parameters*		
open	**Path** /data/jarvis/ text.txt	**Pid** 1184	**Tid** 1184

Table 9.3: new FileInputStream(...) API mapping

System Calls	*Parameters*		
`fork`	**Pid** 1084	**Tid** 1084	**Child ID** 1105
`execve`	**Pid** 1105	**Tid** 1105	**Function** /system/bin/sh
`open`	**Pid** 1105	**Tid** 1105	**Path** `/dev/__properties__`
`open`	**Pid** 1105	**Tid** 1105	**Path** `/dev/tty`

Table 9.4: `Runtime.exec()` API mapping

System Calls	*Parameters*					
`open`	**Path** `/dev/urandom`	**Pid** 1129	**Tid** 1151			
`open`	**Path** `/data/misc/keychain/pinst`	**Pid** 1129	**Tid** 1151			
`socket`	**Family** AF_LOCAL	**Pid** 1129	**Tid** 1151	**Type** SOCK_STREAM	**Protocol** PF_UNSPEC	**Fd** 35
`clone`	**Pid** 58	**Tid** 389	**Child ID** 1153			
`open`	**Path** `/system/etc/hosts`	**Pid** 1129	**Tid** 1151			
`socket`	**Family** AF_INET	**Pid** 58	**Tid** 1153	**Type** SOCK_DGRAM	**Protocol** PF_PACKET	**Fd** 35
`connect`	**Family** AF_INET	**Pid** 58	**Tid** 1153	**Fd** 35		
`open`	**Path** `/sys/class/net`	**Pid** 1129	**Tid** 1151			

Table 9.5: `OpenHTTPConnection(...)` API mapping

System Calls	*Parameters*		
	Interface	**Code**	**Argument**
`ioctl`	`android.os.IServiceManager`	1	`location`

Table 9.6: `getLastKnownLocation(...)` API mapping

9.5 Toy Sample

Given the filtered mapping, we have to verify if Jarvis is able to log the system calls.

A Toy Sample is a software, an application, a tool that simulates the real one in some particular aspects. It's widely used in the first phase of researches because it ease testing phase and allow to understand the behavior it replicates. In our case, the ToySample simulates workflow of real application that invokes some basic Android services such as sending an SMS, recovering GPS position or open an HTTP connection. These services utilizes the APIs that Jarvis can recognize.

In Figure 9.2 we describe the general structure of the ToySample[3].

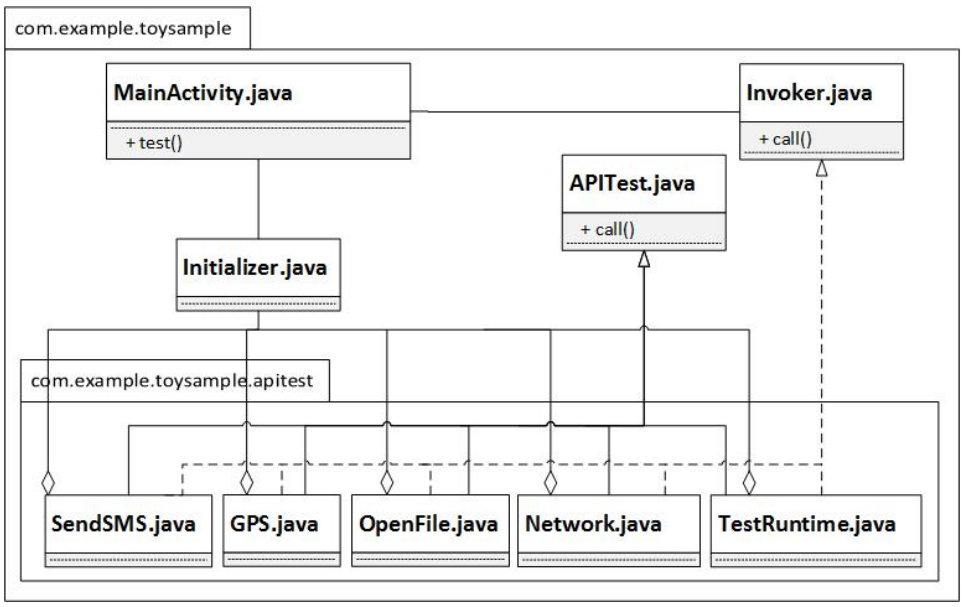

Figure 9.2: Toy Sample Structure

[3]We have not included all the classses that composes the application

ID	Interface	Recognized
1	android.telephony.SmsManager	✓
2	android.location.LocationManager	✓
3	java.net.URL	✓
4	java.io.File	X
5	java.lang.Runtime	X

Table 9.7: Rebuilding Process Result

System Call	Counter		
	Total	*Logged*	*Percentage*
`ioctl`	3518	15	0.42
`open`	298	38	12.75
`access`	210	10	4.76
`all`	4084	121	2.96

Table 9.8: Effectiveness of Filter on Mapped APIs

It is launched directly by Jarvis and this consistitute the unique slightly adaptation the original tool for integration with Toy Sample.

9.5.1 Rebuilding Process

The experiment showed a quite surprising results: Jarvis works better with more elaborate signature with respect to simple APIs. Indeed, in the small set of mapped APIs, Jarvis was capable to recognize those which has either a more robust signature, in terms of variety and number of system calls, or a direct interaction with the Service Manager.

9.5.2 Filter Benchmark

In Table 9.8 we measure the effectiveness of the filter for the system calls `ioctl` and `open` and `access`. Data are related to the execution time of ToySample, which is around 6 seconds.

Although we did not have filters for some system calls, and those implemented are quite rudimentary, they substantially reduce the load on the kernel.

Conclusion

In this final pages we try to take stock and draw conclusion of our work. We describe the strong as well as the weak points of our tool, providing some hints for future improvement and making some research proposal. Our work has been a first trial to:

1. produce a robust documentation for Binder, focusing on the interactions between the driver and the overlying framework;
2. implement a flexible and expandable tool that produce a rich log tracking system calls in a smart way;
3. develop a filtering mechanism to reduce overload, which could allow testing and using on real environment;
4. verify the goodness of approach.

The results show that:

- It's impossible to recognize high-level APIs with a small signature, at worst with only a system call, such as `Runtime.exec` and `FileInputStream`, because their low-level representation is not disjoint enough to associate that pattern to a specific high-level behaviour.
- The key system calls are:
  - `ioctl`, because is related to Binder so to data exchanged between processes and to remote procedure call of key services by activities;
  - `open`, because is related to file and permits to check how Android use its storage;
  - `access`, because is related to permissions and allows monitoring the security control at low-level.
- Filtering mechanism is fundamental to clean the log from useless and meaningless elements and it might constitues the chance to test Jarvis into device devoted to real uses.

- Binder protocol makes really easy, for a tool like Jarvis, monitoring the system manager because the name of required service is easily recoverable by the data buffer.

Open Issues

Jarvis is designed to be flexible and expandable, leaving to future developers the faculty to log new system calls without modification to existent code. We try to suggest some improvements that, for several reasons, have not been implemented yet. These modifications concern both adding new capabilities and refining upon testing environment:

APIs Database

Actually Jarvis saves the high-level log in a file. It is reasonable, increasing the number of mapped APIs, to create a SQLlite Database that may be easily and fast queried, and to better oranized the rebuilder component in a hierarchical way adding, for instance, a cache mechanism.

Dynamic Filtering

The filter is externally provided to Jarvis by a file on which each system call has its own syntax. Jarvis kernel module provides also other keyword-based filter mechanism that use the process id, the thread id, and the file descriptor. It could be implemented a filter that changes its behavior at runtime. For instance, we might be interesting to monitor only the applications that require a specific service to `ServiceManager`. Jarvis driver could record all processes (by means of their IDs) that makes a determined operation, in this case a request to `ServiceManager`, simply adding dynamically an element to the filter list.

Real Devices

Filter mechanism turned out to be a good intuition, the overload for the kernel in terms of memory space and time dedicated to logging operation is drastically reduced. It could allow using the tool on rooted real devices, in order to check the behavior of Jarvis during the normal utilization of the mobile device.

Appendices

Binder Terminology

Summary of Binder terminology [5]:

Binder (Framework) : The overall IPC architecture;

Binder Driver: The kernel-level driver that manages the communication across process boundaries;

Binder Protocol: Low-level protocol (`ioctl`-based) used to communicate with the Binder driver;

Binder Interface (a.k.a iBinder): A well-defined behavior (i.e. methods) that Binder Objects must implement;

AIDL: Android Interface Definition Language used to describe methods signatures on a IBinder Interface;

Binder (Object): A generic implementation of the IBinder interface;

Binder Token: An abstract 32-bit integer value that uniquely identifies a Binder object across all processes on the system;

Binder Service: An actual implementation of the Binder (Object) that implements some methods;

Binder Client: An object exploiting the behavior offered by a binder service;

Binder Transaction: An act of invoking an operation (i.e. a method) on a remote Binder object, which may involve sending/receiving data, over the Binder Protocol;

Parcel: Container serializing messages (data and object references) sent using Binder. A unit of transactional data - one for the outbound request, and another for the inbound reply;

Marshalling: A procedure for converting higher level applications data structures (i.e. request/response parameters) into parcels for the purposes of embedding them into Binder transactions;

Unmarshalling: A procedure for reconstructing higher-level application data-structures (i.e. request/response parameters) from parcels received through Binder transactions;

Proxy: An implementation of the AIDL interface that(un) marshals data and maps method calls to transactions submitted via a wrapped IBinder reference to the Binder object;

Stub: A partial implementation of the AIDL interface that maps transactions to Binder Service method calls while (un)marshalling data

Context Manager (a.k.a. servicemanager): A special Binder Object that is used as a registry/lookup service for other Binder Objects (name → handle mapping).

Bibliography

[1] Thomas Ball. "The concept of Dynamic Analysis". In: *Bell Laboratories* (1999).

[2] T. Holz C. Willems and F. Freiling. "Toward automated dynamic malware analysis using cwsandbox." In: *Proc. of the IEEE Symposium on Security and Privacy* (2007).

[3] Wietse Venema Dan Farmer. *Forensic Discovery*. Ed. by Addison Wesley. 2004.

[4] Christopher Kruegel Darren Mutz Fredrik Valeur and Giovanni Vigna. "Anomalous System Call Detection". In: *Tissec* (2006).

[5] Saketh Paranjape Dhinakaran Pandiyan. *Android Architecture and Binder*. `http://rts.lab.asu.edu/web_438/project_final/Talk8AndroidArc_Binder.pdf`. 2012.

[6] DHS and FBI. "Threats of Mobile Devices Using the Android Operating System". In: *Unclassified* (2013).

[7] Aristide Fattori et al. "CopperDroid: On the Reconstruction of Android Malware Behaviors". In: MA-2014-01 (Feb. 2014).

[8] Google. *Android Documentation*. `http://developer.android.com/reference/packages.html`. 2014.

[9] Google. *Android Kernel 3.10 - Binder driver source code*. `https://android.googlesource.com/kernel/common.git/+/android-3.10/drivers/staging/android/binder.c`. 2014.

[10] Google. *Android Kernel 3.4 - Binder driver source code*. `https://android.googlesource.com/kernel/common.git/+/android-3.4/drivers/staging/android/binder.c`. 2013.

[11] Google. *Android Security Overview*. `https://source.android.com/devices/tech/security/`. 2014.

[12] Google. *Goldfish Project: kernel sources for the emulated platform*. `https://android.googlesource.com/kernel/goldfish.git`. 2011.

[13] P.R.L. Eswari Grandhi Jyostna Pareek Himanshu. "Detecting Anomalous Application Behaviors using a System Call Method over Critical Resources". In: *Advances in Network Security and Applications: 4th International Conference* (2011).

[14] Palmsource Inc. *Open Binder documentation*. `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/BinderIPCMechanism.html`. 2005.

[15] Andrew Honig Michael Sikorski. *Practical Malware Analysis - The Hands-On guide to Dissecting Malicious Software*. Ed. by No Starch Press. 2012.

[16] Oracle. *Java Platform Documentation - Standard Edition*. `http://docs.oracle.com/javase/8/`. 2014.

[17] T. Rosa. *Android Binder Security Note: On Passing Binder Through Another Binder*. 2011.

[18] Thorsten Schreiber. "Android Binder". MA thesis. Rhus-Universitat Bochum, 2011.

[19] Stephanie Forrest Steven A. Hofmeyr and Anil Somayaji. "Intrusion Detection using Sequences of System Calls". In: *Journal of Computer Security* 6 (1998), pp. 151–180.

[20] Antonio Bianchi Tommaso Latini Yuri Iozzelli and Yanick Frantantonio. *Jarvis*. `https://git.seclab.cs.ucsb.edu/gitlab/antoniob/androidbinder`. 2014.

[21] Avishai Wool. *Packet Filtering and Stateful Firewalls - Handbook of Information Security, vol. III*. Ed. by John Wiley Sons. 2006.

[22] Zhou W. Zhou Y. Wang Z. and Jiang X. "You get off of my market: Detecting Malicious App in official and Alternative Android Markets". In: *Proceedings of the 19th Network and Distributed System Security Symposium* (2012).

List of Figures

List of Tables