Universidad Carlos III de Madrid
Escuela Politécnica Superior

Trabajo de Fin de Grado

Monitorización de Eventos en Android a Nivel Nativo

Autor: Guillermo Izquierdo Moreno
Tutor: Sergio Pastrana Portillo

Grado en Ingeniería Informática

LEGANÉS, MADRID
JUNIO 2016

“Por desgracia, no existe una píldora roja (de la verdad) que puedas tragar con un vaso de agua, como lo hizo Neo (Matrix). Lo que existe es el pensamiento crítico y la persistencia en no aceptar algo sólo porque otros lo dicen o porque es lo que opinan los poderosos, la mayoría, los «otros»… he intentado demostrarte cómo puedes combinar la perseverancia en la búsqueda de la verdad con el pensamiento crítico para poder discernir las realidades básicas, y a menudo tristes, de nuestro entorno.

Sin duda muchas veces te arrepentirás de no haber tomado la píldora azul (del conformismo). Pero también habrá momentos, siempre y cuando decidas tomar la píldora de la verdad amarga, en los que desenmascararás las mentiras de los poderosos, revelando su fealdad y su estupidez. Ésta será tu recompensa.”

Yanis Varoufakis

Agradecimientos

Con la entrega de este documento culmina un periodo. Han sido cuatro largos años de esfuerzo y dedicación en los que he pasado por muchas situaciones adversas, como la reducción de la vida social, el déficit de sueño, la intranquilidad constante, etc. Pero no todo ha sido negativo, durante mi estancia en la universidad he podido conocer a gente fantástica y he vivido una evolución personal de la que estoy orgulloso.

Me gustaría dedicar este proyecto a una serie de personas.

En primer lugar, quiero agradecer la labor del personal de la Educación Pública, pues son los que llevan a hombros nuestro sistema educativo y se esfuerzan mucho por mantener el nivel pese a las adversidades. De todos, para todos.

A mi tutor Sergio, por su ayuda y dedicación. Él me ha permitido conocer al personal del laboratorio del grupo COSEC, el cual me ha tratado en todo momento como a uno de los suyos.

A mis amigos de toda la vida, y a los que nos hemos encontrado por el camino. Cristian, Jorge, Coca, Alex, Nacho, Díaz, Crítiko, Pérez y Bifoo. A partir de ahora espero salir más a menudo.

A los amigos que he hecho en el grado. Mario, de Lucía, Álvaro, Saúl, Alex, Sandra, Juanlu, sois geniales. Esto no es un agradecimiento de despedida.

A mis amigos del pueblo. Me paso todo el año deseando ir a veros.

A mis padres, que han permitido que todo esto haya sido posible. Aunque casi no coincidimos en casa y cuando estoy suelo estar encerrado con mis cosas, espero que sepáis que os quiero mucho y que valoro todo lo que hacéis por mí.

A mi hermana, persona con la que he crecido y he compartido todo. Aunque hayas tenido que emigrar al extranjero, cada vez que nos vemos es como si no hubiese pasado el tiempo.

Al resto de mi familia, que me ha apoyado siempre y se ha preocupado por mí.

Por último, quería hacer una especial mención a la persona que ha cargado conmigo durante estos cuatro años. La persona que solo se ha llevado la parte negativa de este grado, mi compañera de viaje, mi desahogo y mi consuelo. Te quiero peque.

Índice de Contenidos

Capítulo 1: Introducción 9
1.1. Motivación 9
1.2. Objetivos 10
1.3. Estructura del texto 11
Capítulo 2: Estado del Arte 12
2.1. Dispositivos Móviles 12
2.2. Android 12
2.3. Visión General de Android 13
2.4. Arquitectura Android 14
2.5. Fundamentos de las Aplicaciones Android 15
2.5.1. APK 15
2.5.2. Componentes 15
2.5.3. Fichero Manifest.xml 17
2.6. Android Binder 17
2.6.1. Terminología Relacionada con Android Binder 18
2.6.2. Modelo de IPC 19
2.6.3. Transacciones 21
2.6.4. Android Interface Definition Language (AIDL) 23
2.7. Memoria Compartida en Android 24
2.7.1. Address space layout randomization (ASLR) 25
2.7.2. Android Shared Memory (ashmem) 25
2.8. Análisis Dinámico de Binder IPC 26
2.8.1. Análisis en Entorno Android Emulado – CopperDroid 26
2.8.2. Análisis en Entorno Android Real 28
2.9. Detección de Entornos de Análisis 30
2.10. Interceptación de Llamadas al Sistema en Entornos Android Reales 31
2.10.1. Hook de Llamadas al Sistema 31
2.10.2. Registro de SELinux 33
2.10.3. Strace 34

Capítulo 3: Análisis 35
3.1. Planteamiento del Problema 35
3.2. Descripción de la solución 35
3.3. Casos de Uso 36
3.3.1. Caso de uso CU-01: Interceptación de Binder Transactions. 36
3.3.2. Caso de uso CU-02: Interceptación de llamadas al sistema regulares. 37
3.4. Especificación de Requisitos. 38
3.4.1. Plantilla de Requisito 38
3.4.2. Requisitos de Usuario. 39
3.4.3. Requisitos de Sistema. 44
Capítulo 4: Diseño 55
4.1. Visión General del Sistema 55
4.2. Definición de la Arquitectura del Sistema 56
4.2.1. Especificación del Entorno Tecnológico 56
4.2.2. Descomposición en Subsistemas de Diseño 57
4.2.3. Identificación y Descripción de los Módulos del Sistema. 59
4.2.4. Diagrama de Despliegue o Distribución 61
4.2.5. Diseño de la Arquitectura de Módulos del sistema 61
4.3. Diseño Físico de Datos 72
4.3.1. Estructura de Paquete 72
4.3.2. Estructura de Mensaje 73
4.3.3. Estructura de Paquetes de Mapeos de un Proceso 75
Capítulo 5: Implementación 76
5.1. Injector Module 76
5.1.1. Injector 76
5.2. Monitor Module 76
5.2.1. Contructor de Libhook 76
5.2.2. Hooker 76
5.2.3. Monitor 77
5.3. Communication Module 80
5.3.1. Componentes de la parte servidor – Extraction APK 81
5.3.2. Componentes de la parte cliente - Libhook 84

5.4. Extraction Module ... 87
5.4.1. Messenger IPC ... 87
5.4.2. Envío de paquetes a la red ... 88
5.5. Data Analysis Module ... 89
5.6. Launcher Module ... 89
Capítulo 6: Verificación, Validación y Experimentación ... 90
6.1. Verificación del Sistema ... 90
6.2. Extracción de Comportamientos ... 92
Capítulo 7: Planificación ... 95
7.1. Ciclo de Vida del Proyecto ... 95
7.2. Diagrama de Gantt ... 96
Capítulo 8: Presupuesto ... 98
8.1. Costes del Proyecto ... 98
8.2. Propuesta de Venta del Sistema ... 100
Capítulo 9: Marco Regulatorio ... 101
9.1. Marco Regulatorio Aplicable al Software Desarrollado ... 101
9.1.1. Tipología del Software Según Su Licenciamiento ... 101
9.1.2. Software de Uso Legal ... 101
9.1.3. Regulación Sobre la Propiedad Intelectual ... 101
9.1.4. Legislación Aplicable al Sistema de Información ... 102
9.1.5. Normativas y Estándares ... 102
9.2. Marco Regulatorio del Uso y Adquisición de Software Heredado ... 102
9.2.1. Tipología del Software Según Su Licenciamiento ... 102
9.2.2. Software de Uso Legal ... 103
9.2.3. Regulación Sobre la Propiedad Intelectual ... 103
Capítulo 10: Conclusiones y Futuros Trabajos ... 104
10.1. Conclusiones ... 104
10.2. Trabajos Futuros ... 106
Anexo 1: Implementación ARMINJECT ... 107
Anexo 2: Abstract ... 109
Bibliografía y Recursos ... 119

Índice de Figuras

Figura 1. Arquitectura de Android 15
Figura 2: Flujo de comunicación a través del Binder [6] 19
Figura 3: Estructura binder_write_read 21
Figura 4: Estructura binder_transaction_data 22
Figura 5: Estructura de datos en el binder 23
Figura 6: Estructura de la memoria física en el Kernel de Linux 24
Figura 7: Algoritmo de *unmarshalling* usado en *Copperdroid* 28
Figura 8: Esquema de monitorización propuesto en [12] 29
Figura 9: Hooking de llamadas al sistema [14] 32
Figura 10: CU-01 36
Figura 11: CU-02 37
Figura 12: Diagrama de distribución 61
Figura 13: Diagrama de componentes 62
Figura 14: Estructura de un paquete 73
Figura 15: Estructura de mensaje 73
Figura 16: Estructura de cabezera de mensaje 73
Figura 17: Estructura cuerpo llamadas regulares 74
Figura 18: Estructura cuerpo llamada ioctl 74
Figura 19: Estructura mensaje de mapeos 75
Figura 20: Estructura entrada 75
Figura 21: Relación de llamadas interceptadas 77
Figura 22: Tipos de ioctl usados por el Binder 79
Figura 23: Código de Binder Transaction 80
Figura 24: Clase de métodos nativos JNI 81
Figura 25: Implementación de los métodos nativos JNI 82
Figura 26: Server Interface del módulo de comunicación 83
Figura 27: Implementación del método getDemoServ 85
Figura 28: Implementación del método shareFD en el cliente 85
Figura 29: Esquema del protocolo de memoria compartida del módulo de comunicación 87
Figura 30: Envío de paquetes por la red 88
Figura 31: Eventos monitorizados para la app Google Chrome 92
Figura 32: Evolución de eventos monitorizados para la app Google Chrome 93
Figura 33: Eventos monitorizados para la app de la calculadora 94
Figura 34: Evolución de los eventos monitorizados para la app de la calculadora 94
Figura 35: Ciclo de vida del proyecto 96
Figura 36: Planificación temporal del proyecto (Gantt) 97

Índice de Tablas

Tabla 1: Plantilla de Requisito 38
Tabla 2: CAP-01 39
Tabla 3: CAP-02 39
Tabla 4: CAP-03 40
Tabla 5: CAP-04 40
Tabla 6: CAP-05 40
Tabla 7: CAP-06 41
Tabla 8: CAP-07 41
Tabla 9: RES-08 41
Tabla 10: RES-09 42
Tabla 11: RES-10 42
Tabla 12: RES-11 42
Tabla 13: RES-12 43
Tabla 14: RES-13 43
Tabla 15: RES-14 43
Tabla 16: RES-15 44
Tabla 17: RES-16 44
Tabla 18: RF-01 45
Tabla 19: RF-02 45
Tabla 20: RF-03 46
Tabla 21: RF-04 46
Tabla 22: RF-05 46
Tabla 23: RF-06 47
Tabla 24: RF-07 47
Tabla 25: RF-08 48
Tabla 26: RF-09 48
Tabla 27: RF-10 49
Tabla 28: RF-11 49
Tabla 29: RF-12 49
Tabla 30: RN-01 50
Tabla 31: RN-02 50
Tabla 32: RNF-01 51
Tabla 33: RNF-02 51
Tabla 34: RNF-03 51
Tabla 35: RNF-04 52
Tabla 36: RNF-05 52
Tabla 37: RNF-06 52

Tabla 38: RNF-07 53
Tabla 39: RNF-8 53
Tabla 40: RNF-09 53
Tabla 41: RNF-10 54
Tabla 42: Entrono Tecnológico 56
Tabla 43: Subsistema Injector Module 57
Tabla 44: Subsistema Libhook 57
Tabla 45: Subsistema Extraction APK 58
Tabla 46: Subsistema Data Analysis Module 58
Tabla 47: DEP-01 63
Tabla 48: DEP-02 63
Tabla 49: DEP-03 63
Tabla 50: DEP-04 63
Tabla 51: DEP-05 64
Tabla 52: DEP-06 64
Tabla 53: DEP-06 64
Tabla 54: DEP-07 64
Tabla 55: DEP-08 65
Tabla 56: DEP-09 65
Tabla 57: DEP-10 65
Tabla 58: CO-01 66
Tabla 59: CO-02 66
Tabla 60: CO-03 67
Tabla 61: CO-04 68
Tabla 62: CO-05 69
Tabla 63: CO-06 69
Tabla 64: CO-07 70
Tabla 65: CO-08 70
Tabla 66: CO-09 71
Tabla 67: CO-10 71
Tabla 68: CO-11 72
Tabla 69: PV-01 90
Tabla 70: PV-02 91
Tabla 71: PV-03 91
Tabla 72: Información del Proyecto 98
Tabla 73: Personal del Proyecto 98
Tabla 74: Equipos 99
Tabla 75: Otros Costes Directos del Proyecto 99
Tabla 76: Resumen de Costes 100
Tabla 77: Propuesta de Venta 100
Tabla 78: Mantenimiento Anual 100

Capítulo 1: Introducción

1.1. Motivación

El panorama del análisis de las aplicaciones Android es un campo que se está tratando mucho en los últimos años. Esto es debido a que Android es el sistema operativo móvil más popular del mercado, y por tanto en el que más piezas de malware se distribuyen. Entre otros motivos del auge de malware en Android, se encuentra que es un proyecto *open source*, por lo que su código está expuesto a ser inspeccionado en busca de vulnerabilidades.

Las distintas investigaciones sobre este ámbito parten de una premisa clara: es necesario el análisis dinámico de las aplicaciones, que pueda complementar análisis estático. El malware se incluye a menudo en códigos ofuscados impracticables en un análisis estático único. Además, el rastro que deja el malware tras ser ejecutado en un sistema a menudo no proporciona una cantidad de datos significativa para conocer su comportamiento, es necesario recopilar información durante la ejecución del malware, pues esta es la única forma de recuperar valores almacenados en memoria, comunicación entre procesos, conexiones de red, etc.

Junto a estas investigaciones han aparecido diferentes herramientas de análisis dinámico de comportamientos de malware las cuales han tomado diferentes formas. Algunas están basadas en entornos emulados o virtualizados, como *CopperDroid*, *DroidScope* o *Google Bouncer* (el cuál analiza aplicaciones subidas al mercado oficial de Android, Google Play Store). Otras propuestas están basadas en plataformas reales, como *BareDroid*, pero el objetivo que persiguen todas es el mismo: conseguir una monitorización de las aplicaciones completa de forma transparente a las mismas.

Pese a las diferencias de su arquitectura, el planteamiento de la monitorización pasa en todos ellos por la interceptación de llamadas al sistema regulares y de la comunicación entre proceso, que en Android se realiza en su mayoría a través del Binder, en el núcleo del sistema operativo.

Al existir un gran número de investigaciones y herramientas en este campo, la mejora de la técnica es continua, razón por la que cada vez se elaboran herramientas más sofisticadas. Esta sofisticación del sistema es importante, porque a medida que avanza la investigación sobre el análisis de comportamiento, también se refina la técnica de evasión del malware. Una muestra de malware puede detectar si se encuentra en un entorno de análisis e interrumpir el flujo de actividades maliciosas. Esta circunstancia no es completamente negativa, pues el estudio del comportamiento evasivo también es uno de los objetivos de este campo.

La construcción de una herramienta dedicada a la monitorización y análisis de los comportamientos de aplicaciones Android supone, en primer lugar un reto, pues es necesario comprender los cimientos del sistema operativo, y en segundo lugar la posibilidad de realizar un proyecto innovador, con una utilidad real, que pueda contribuir de alguna forma a la comunidad científica.

Lo positivo de trabajar sobre un sistema operativo Android, es que al estar dedicado en exclusivo a los dispositivos móviles, tiene una arquitectura menos compleja que la que podría tener un ordenador de sobremesa. Esto, combinado con la disponibilidad de su código y la extensa documentación al respecto facilitan la construcción de la herramienta de forma considerable. De forma paralela, la complejidad técnica del desarrollo del proyecto, donde todas las capas de la arquitectura se ven implicadas, proporcionará un conocimiento completo del funcionamiento del sistema operativo Android a todos los niveles.

1.2. Objetivos

El objetivo final de este proyecto consiste en desarrollar un sistema de análisis dinámico del comportamiento de aplicaciones Android. Esto pasa por la interceptación de llamadas al sistema y de la comunicación entre procesos relativos a una aplicación Android.

El sistema estará basado en una plataforma real, sobre la que se ejecuta el sistema operativo Android, y estará dividido en una parte residente en el dispositivo, dedicada a la monitorización de la aplicación y otra parte residente en un servidor externo, que se ocupe del análisis de los datos extraídos en la monitorización. Será necesaria algún tipo de comunicación entre las partes.

Los sub-objetivos del proyecto son los siguientes:

- Analizar el panorama actual de herramientas de monitorización y de análisis dinámico para Android.
- Proporcionar una herramienta que permita monitorizar de forma transparente las llamadas al sistema y la comunicación entre procesos de una aplicación Android en ejecución.
- Realizar una monitorización transparente a la aplicación.
- Conseguir extraer patrones de comportamiento de aplicaciones a través de los datos interceptados por la herramienta.

1.3. Estructura del texto

Este documento recopila el trabajo realizado durante el proceso de desarrollo del Sistema final, incluyendo el problema analizado, el diseño de la solución, la implementación del sistema final y la evaluación del mismo. En este documento también se describe el panorama del análisis dinámico de comportamientos en Android en función de las investigaciones más trascendentales que se han realizado hasta la fecha. Como es habitual en todo proyecto, se incluye la planificación, el cálculo de costes que ha supuesto el desarrollo completo del proyecto y la legislación y normativas que se le aplican.

La estructura del documento se describe a continuación.

- Capítulo 1: Introducción: incluye el contexto o motivación, objetivos del proyecto y la estructura del documento.
- Capítulo 2: Estado del Arte: estudio de las principales líneas de investigación relacionadas así como de las herramientas similares y de las técnicas que utilizan.
- Capítulo 3: Análisis: estudio del problema e identificación de los requisitos que el sistema final debe satisfacer.
- Capítulo 4: Diseño: ofrece una visión general de la arquitectura del sistema, así como la especificación técnica de los subsistemas de diseño, sus componentes y el modelo físico de los datos.
- Capítulo 5: Implementación: describe los detalles más significativos de la implementación del sistema.
- Capítulo 6: Verificación, Validación y Evaluación: conjunto de pruebas realizadas para corroborar el cumplimiento de los requisitos del sistema y demostración de posible uso.
- Capítulo 7: Planificación: descripción del ciclo de vida del proyecto y de la planificación seguida durante el proceso de desarrollo del mismo.
- Capítulo 8: Presupuesto: desglose de los costes que ha supuesto el proyecto y propuesta de venta del mismo.
- Capítulo 9: Marco Regulatorio: legislación, normativa y estandarización aplicable al sistema.
- Capítulo 10: Conclusiones y Futuros Trabajos:

Al final del documento se incluyen anexos con información adicional sobre alguna característica del sistema y el resumen del proyecto en lengua inglesa. Los recursos bibliográficos referenciados a lo largo del documento se encuentran en la última sección.

Capítulo 2: Estado del Arte

2.1. Dispositivos Móviles

En la actualidad, a pesar de la amplia definición que abarca el concepto de dispositivo móvil, este término normalmente se refiere a teléfonos inteligentes (*smartphone*) o tabletas (*tablets*).

Tanto la versatilidad de estos dispositivos como la posibilidad que ofrecen de comunicarse a través de redes móviles (tales como Wifi o 3G, incluso 4G en el último periodo), y una máxima portabilidad, los han convertido en los reyes del mercado tecnológico. Muchos fabricantes han invertido grandes cantidades de dinero con el objetivo de desarrollar productos incluso más potentes y con más características para poder atraer más usuarios.

De forma paralela a la aparición de este tipo de tecnología, han surgido grandes compañías centradas en el desarrollo de sistemas operativos para dispositivos móviles. Los sistemas operativos móviles más populares actualmente son Android OS, desarrollado por Android Inc. y adquirido por Google más tarde, y iOS, de Apple Inc.

2.2. Android

Android es indudablemente el sistema operativo móvil más exitoso del mercado. El número de dispositivos que incorpora el sistema operativo Android creció exponencialmente desde que se lanzó a la venta el primer terminal con este sistema operativo en octubre de 2008, el HTC Dream. La razón por la que Android se ha convertido en el sistema operativo más utilizado es porque fue diseñado por el objetivo de ser instalado en numerosos dispositivos y modelos. Esto ha causado a su vez un alto grado de fragmentación entre sus distribuciones.

Android cuenta con su propio distribuidor de aplicaciones, Google Play. Los desarrolladores publican aquí sus aplicaciones, las cuales están disponible para su descarga gratuita o su compra. Así como en otras plataformas, la publicación de aplicaciones está regulada por una serie de pruebas impuestas por la compañía. Su objetivo es evaluar los contenidos, el rendimiento y la seguridad de las aplicaciones. Además, Android permite instalar aplicaciones de fuentes externas a Google sin necesidad de hacer cambios sobre el sistema operativo, como pasa en iOS.

La libertad para desarrollar e instalar aplicaciones, sumada al crecimiento de mercados alternativos, han posibilitado la aparición de una gran comunidad de desarrolladores y, en consecuencia, de una extraordinaria cantidad de aplicaciones.

Esta situación también ha sido provocada por la política de instalación de aplicaciones de Android. El sistema permite la instalación de aplicaciones firmadas, pero acepta las auto-firmadas por cualquier autor, no exige la acreditación de una autoridad certificadora. Además,

las actualizaciones y la integridad de una aplicación dependen únicamente de la honestidad del autor. Esto le ha convertido en el sistema operativo móvil donde se distribuyen más muestras de malware, por la facilidad de distribuir aplicaciones maliciosas que los usuarios instalarán directamente en sus dispositivos, como por ejemplo troyanos en forma de aplicaciones reempaquetadas.

2.3. Visión General de Android

Android es un sistema operativo móvil basado en un modelo simplificado del núcleo (*Kernel*) de Linux al que se le han añadido extensiones específicas para las necesidades móviles.

Toda la implementación del sistema se apoya en cuatro lenguajes de programación: Java, C++, C y ensamblador. Estos lenguajes se pueden encontrar a lo largo del sistema en diferentes capas de abstracción, las cuales se corresponden con las características que ofrece cada uno. Java se usa a nivel de aplicación, C++ se usa como compuerta a la funcionalidad del *Kernel* en aplicaciones nativas y bibliotecas de Android, mientras que C y ensamblador se utilizan principalmente en el *Kernel*.

Dado que Android OS está orientado a la interacción con el usuario, la mayoría de sus programas toman la forma de aplicaciones auto-contenidas formadas por uno o varios componentes donde se incluyen interfaces de usuario (UI).

Estas aplicaciones se ejecutan en un entorno controlado. La *Dalvik Virtual Machine* (DVM) es un entorno de ejecución aislado donde las aplicaciones implementadas en Java son procesadas en sus propios *bytecodes* y ejecutadas, de forma similar a como actúa la *Java Virtual Machine* (JVM). La máquina virtual tiene una arquitectura de registros altamente compatible con la arquitectura ARM sobre la que se ejecuta, razón que le permite tener un alto rendimiento.

Hay una diferencia entre los programas compilados para la máquina virtual y los programas compilados para ejecutarse en una plataforma específica, como Intel x86 o ARM. Los compilados para estas plataformas se denominan nativos y se pueden ejecutar directamente, sin pasar por la máquina virtual, a diferencia de Java.

Cada aplicación de Android se ejecuta como una instancia de la DVM en su propio proceso de espacio de usuario. Estos procesos están aislados en una *sandbox*, donde se aplica una política de privilegio mínimo por razones de seguridad. Una aplicación únicamente puede tener acceso a utilidades del sistema o a componentes de terceros si los solicita explícitamente los permisos correspondientes en su fichero *Manifest.xml* en el momento de instalación [1]. La forma que tiene una aplicación de acceder a los mecanismos del sistema operativo de bajo nivel, como las llamadas al *Kernel*, es a través de la *Java Native Interface* (JNI), que permite al código Java ejecutar código compilado en bibliotecas escritas en otros lenguajes (por ejemplo, C++) haciendo uso de una API bien definidas [2].

Existen distintos componentes de las aplicaciones de Android (por ejemplo, actividades, servicios, etc.), los cuales llevan a cabo distintas tareas y pueden ser invocados por el sistema operativo o directamente por el usuario. Cuando diferentes componentes tienen que intercambiar datos o realizar acciones en otros componentes, la comunicación entre componentes se lleva a cabo haciendo uso de los *Intent*, que son mensajes asíncronos que contiene un URI que identifica un componente de aplicación y una acción que identifica la operación que se solicita. No es necesario que el componente sea activado desde la propia aplicación, su activación puede ser el resultado de una comunicación entre procesos [2].

Android hereda el módulo de seguridad *Security-Enhanced Linux* (SELinux) para el *Kernel* de Linux, que proporciona el mecanismo para establecer políticas de seguridad para el control de acceso de las aplicaciones. En Android se utiliza para definir los límites del aislamiento de aplicaciones en *sandboxes*. Sus políticas admiten distintos grados, pero a no ser que sea modificado, se aplica el modo no permisivo, por el que las acciones que se consideran ilegítimas están bloqueadas, pero se puede habilitar mayores grados de permisividad. [3]

2.4. Arquitectura Android

El sistema operativo Android está basado en una arquitectura de múltiples capas, mostrada en la Figura 1. Cada capa corresponde a un nivel de abstracción.

La capa de aplicación se sitúa en la parte superior de la arquitectura. Aquí es donde coexisten diferentes aplicaciones. Algunas aplicaciones son herramientas básicas del sistema operativo, como las que ofrece Google (Maps, Contacts, etc.), pero se pueden instalar aplicaciones externas. Todas ellas deben ser implementadas en Java y distribuidas como *Android Packages Archive* (APK).

La siguiente capa es el marco de aplicaciones. Los desarrolladores de Android acceden a esta capa a través de APIs bien definidas. En este nivel es donde una aplicación puede ofrecer y consumir servicios de otra, incluyendo los servicios del sistema Android. Esta es la forma que tienen de gestionar la interfaz de usuario (IU), la comunicación entre componentes, etc.

La tercera capa está compartida por las bibliotecas del sistema y el entorno de ejecución de Android. Las bibliotecas del sistema se implementan en código nativo, es decir C y C++, y dan soporte completo a la funcionalidad de la capa superior, pero a bajo nivel. El entorno de ejecución de Android está compuesto por la máquina virtual donde se ejecutan las aplicaciones, la *Dalvik Virtual Machine*, y un conjunto de bibliotecas del núcleo dan soporte a este entorno de ejecución. Estas son las bibliotecas específicas de Dalvik, las bibliotecas de interoperabilidad de Java y las bibliotecas de desarrollo de Android.

El nivel de abstracción más bajo corresponde al núcleo de Linux, que soporta la comunicación entre procesos, la planificación de los mismos y la gestión del hardware, como la memoria,

procesador, periféricos, formas de almacenamiento, etc. Su funcionalidad tiene que ser requerida mediante llamadas al sistema. [4]

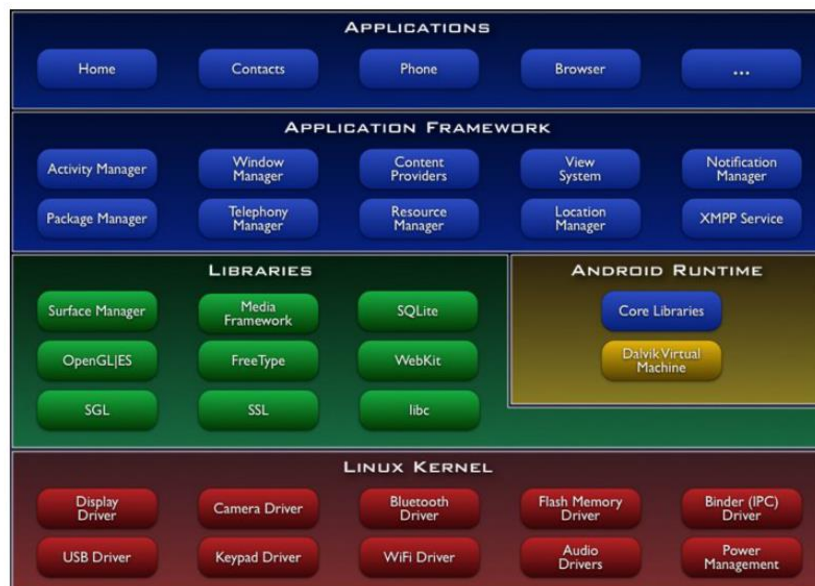


Figura 1. Arquitectura de Android

2.5. Fundamentos de las Aplicaciones Android

2.5.1. APK

Android Application Package es el tipo de fichero que utiliza Android en la distribución e instalación de aplicaciones móviles y middleware. Contiene toda la funcionalidad de la aplicación pre-procesada en *bytecodes* (fichero *.dex*) interpretables por la DVM. También contiene los ficheros de la aplicación relacionados con la firma, permisos, recursos, etc. Las aplicaciones Android son considerados como autónomas.

2.5.2. Componentes

Android es un sistema complejo que consta de varios componentes. En esta sección se va a realizar una breve introducción a aquellos que tienen una relevancia para el entendimiento de este Trabajo de Fin de Grado, ya que han sido de alguna forma utilizados durante el desarrollo.

2.5.2.1. *Activity*

La clase *Activity* representa una tarea o acción que el usuario puede hacer. Casi todas las actividades interactúan con el usuario, por lo que tienen asociada una interfaz de usuario (UI). Las actividades en ejecución se apilan en el sistema, donde la actividad en la parte superior de

la pila es sobre la que se hace foco. Una actividad puede aparecer en tres estados mientras esta activa: *resumed* o *running*, cuando el usuario hace foco en ella, *paused*, cuando todavía está asociada al gestor de ventanas (pantalla), pero el usuario no le hace foco, y *stopped* cuando no está asociada al gestor de ventanas. En esos tres estados, la actividad se mantiene en memoria donde conserva toda la información relativa a su estado.

Si una actividad está en *paused* o *stopped*, el sistema puede acabar con ella si se requiere su espacio de memoria.

2.5.2.2. *Service*

La clase *Service* representa un componente de aplicación que puede llevar a cabo operaciones de larga duración sin interactuar con el usuario. Se usan para suministrar algún tipo de funcionalidad a la propia aplicación o a terceras.

Un servicio se ejecuta en el hilo principal del proceso que lo instancia. En caso de un trabajo intensivo de la CPU o de operaciones de bloqueo, tales como operaciones de entrada y salida (E/S), es muy recomendable crear otro hilo dentro del servicio que se ocupe de ese trabajo.

Un servicio puede adoptar dos formas: *started*, cuando el servicio se instancia invocando *startService()* y se ejecuta en segundo plano de forma indefinida, y *bound* cuando el servicio es enlazado invocando *bindService()* y ofrece una interfaz entre cliente y servidor para permitir a otros componentes interactuar con el servicio a través de comunicación entre procesos (IPC).

Un *IntentService* es una extensión de la clase *Service* que se activan mediante un objeto *Intent*. Este tipo de servicio se ejecuta de forma automática en un hilo y puede estar ejecutando durante el tiempo sin que sea necesario bloquear el proceso principal de la aplicación. A diferencia de *Service,* este componente no puede atender múltiples peticiones simultáneas.

2.5.2.3. *Parcel*

Un objeto de tipo *Parcel* es un contenedor que se puede enviar a través del Android Binder en una comunicación entre procesos. Admite datos y referencias a otros objetos, pero tienen que ser objetos “serializables”, es decir, que implementen la clase *Serializable*, de modo que se puedan codificar dentro de un *Parcel*. El proceso de codificación de un tipo de dato dentro de un *Parcel* se llama *marshalling* y la decodificación *unmarshalling*. Esta clase está implementada como un mecanismo de transporte en IPC de alto rendimiento.

2.5.2.4. *Intent*

Un objeto de tipo *Intent* es un mensaje asíncrono que contiene un URI que identifica de forma exclusiva a un componente de una aplicación y una acción que identifica la operación que se

ejecutará en este componente. Se utilizan básicamente de tres maneras: para iniciar una actividad, para iniciar un servicio y para enviar un mensaje de difusión. Puede incluir una carga de datos que se adjunta en el envío.

2.5.3. Fichero Manifest.xml

Antes de que el sistema pueda iniciar un componente propio de una aplicación, debe saber que existe dicho componente mediante la lectura del fichero *Manifest.xml* de la aplicación. La aplicación debe declarar todos los componentes que va a utilizar en este archivo ya sean propios o ajenos a la misma.

Este fichero también debe identificar los permisos del sistema que se requieren (por ejemplo, acceso a la red), declarar el nivel mínimo de la API de Android requerida y las características software y hardware que requiere (por ejemplo, Bluetooth). Esta declaración se realiza con un formato específico dentro del fichero, el cual tiene que estar ubicado en el directorio raíz del proyecto de la aplicación.

En el momento que la aplicación requiera un componente, el sistema va permitir o denegar el acceso en función de la declaración en su fichero *Manifest.xml* en el momento de instalación. [5]

2.6. Android Binder

Uno de los módulos que permanece en esta versión modificada del núcleo de Linux es el Binder, que ofrece un mecanismo de IPC para el sistema operativo Android además de los ya conocidos, tales como señales, tuberías, colas de mensajes, semáforos y memoria compartida.

El Binder, además de ser un mecanismo de comunicación entre procesos (*Inter Process Communication,* o IPC) específico de Android, es un sistema de invocación de métodos remoto (*Remote Procedure Call* o RPC). Un proceso de Android puede llamar a una rutina en otro proceso Android utilizando el Binder para identificar el método a invocar y pasar los argumentos entre los procesos.

Las utilidades que ofrece el sistema Android a las aplicaciones de usuario a menudo ejecutan como funciones de servicios remotos que se pueden invocar como si fuesen locales.

La *Android Interface Definition Language* (AIDL) es la encargada de resolver estas cuestiones a alto nivel, ya que permite definir un lenguaje a alto nivel de las interfaces a exponer entre servicios que internamente serán mapeadas a clases cliente y servidor (ver Sección 2.6.4). Así, una aplicación no necesita saber si un servicio se está ejecutando en un proceso remoto o en su propio proceso. Esto hace sencillo para un desarrollador, el hecho de exportar servicios a otras aplicaciones Android sin tener que modificar la implementación del servicio.

Un objeto que se puede invocar de forma remota se instancia como un objeto de tipo Binder. Cada objeto Binder es identificable de forma unívoca por un *token* que puede ser compartido entre procesos. Este identificador permite conseguir una referencia al servicio incluso cuando no está publicado por el *Service Manager* siendo solo accesible por aquellos procesos que comparten el *token,* además de por el sistema. Por consiguiente, el Binder puede ser utilizado como un sistema de acceso seguro que depende del conocimiento de ese *token*.

Otra característica de seguridad es que un proceso destinatario de la llamada puede identificar al proceso que le invoca, pues en la petición se incorpora el identificador de proceso UID y PID. Esta característica, combinada con el modelo de seguridad de Android, permite la identificación de procesos.

El Binder ofrece un mecanismo de comunicación entre procesos muy optimizado y completo. Incluso se puede utilizar para establecer una región de memoria compartida entre procesos. [2]

2.6.1. Terminología Relacionada con Android Binder

- *Binder Framework*: la arquitectura IPC que habilita el Android Binder.
- *Binder Kernel Driver*: el *driver* que trabaja a nivel de Kernel para resolver las invocaciones remotas entres procesos, debido a las limitaciones de los mismos en espacio de usuario.
- *Binder Protocol*: el protocolo a bajo nivel, basado en llamadas al sistema de tipo *ioctl*, que se usa para comunicarse con el Binder Driver.
- *Binder Object* (*Binder*): clase base que referencia a un objeto remoto y hace posible su invocación como si fuese local. No es necesario implementar esta clase directamente pues la AIDL provee una interfaz de abstracción.
- *Binder Interface* (*IBinder*): interfaz que describe el protocolo para interactuar con un objeto remoto. Es implementada por la clase *Binder*.
- *Binder Token*: un valor entero de 32 bits que identifica de forma unívoca a un objeto *Binder* entre todos los procesos del sistema.
- *Binder Service*: es una implementación de *Binder Object* que ofrece una funcionalidad que se puede invocar desde otro proceso.
- *Binder Transaction*: es la acción de invocar una funcionalidad en un *Binder Object* (por ejemplo, un método), la cual conlleva el envío de una petición a través del *Binder Kernel Driver*.
- *Parcel*: contenedor de un mensaje y de referencias a otros objetos que se pueden enviar a través de una Binder Interface.
- *Marshalling*: el proceso de codificar estructuras de datos de alto nivel (por ejemplo, los parámetros de las peticiones) en objetos de tipo *Parcel* con el objetivo de incluirlos en las *Binder Transactions*.
- *Unmarshalling*: el proceso de reconstrucción de estructuras de datos de alto nivel a partir de los objetos de tipos *Parcel* recibidos a través de las *Binder Transactions*.

- *Common Interface*: es la declaración de métodos remotos de un *Binder Service*, y es conocida por cliente y servicio. Esta interfaz es abstraída por la AIDL.
- *Proxy* (*Client Interface*): es la implementación de la interfaz del cliente encargada de hacer el un/marshalling de los datos y mapear las llamadas de los métodos remotos para iniciar transacciones a través del *Binder Driver*. Esta interfaz es abstraída por la AIDL.
- *Stub* (*Service Interface*): es la implementación de la interfaz del servicio encargada de mapear las transacciones realizadas sobre los métodos del *Binder Service* y de realizar el un/marshalling de los datos. Esta interfaz es abstraída por la AIDL.
- AIDL: consiste en la interfaz de abstracción de alto nivel de todos los procesos y objetos relacionados con la comunicación a través del Android Binder. Los clientes y servicios de alto nivel pueden delegar todo el procedimiento en la AIDL.
- *Context Manager* (*ServiceManager*): es un *Binder Object* especial cuyo controlador tiene una referencia conocida, y se usa para registrar e identificar servicios de otros *Binder Objects*. A través del nombre del servicio, mapea y devuelve la referencia a este. [6]

2.6.2. Modelo de IPC

Los servicios registrados en el sistema Android se ejecutan en sus propios procesos, por lo que un cliente tiene que utilizar IPC para solicitar su funcionalidad. La política de mínimo privilegio y la ejecución del proceso de forma aislada en su *sandbox* hacen que el Binder sea el principal mecanismo de IPC en Android.

El *Binder Driver* resuelve las *Binder Transactions* a nivel del núcleo. Este copia de datos desde el espacio de memoria de un proceso al de otro y mantiene la correspondencia entre los manejadores de los servicios y los objetos de un proceso específico.

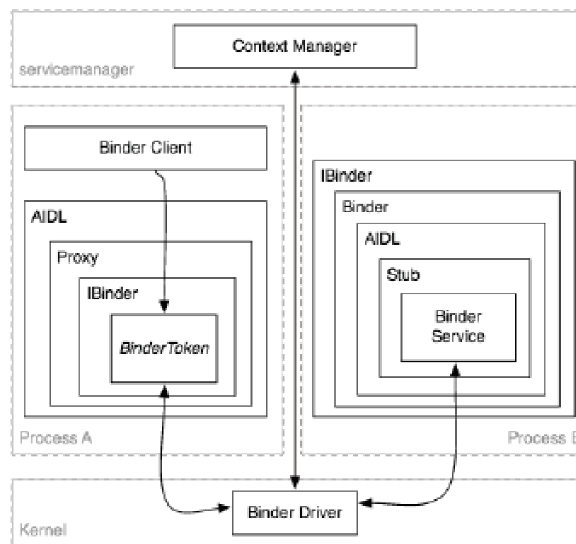


Figura 2: Flujo de comunicación a través del Binder [6]

La Figura 2 muestra una abstracción del flujo de la comunicación a través del *Binder Driver*. Todos los niveles de la arquitectura de Android están involucrados en este modelo de comunicación. Cada nivel proporciona una capa de abstracción de forma que el desarrollador no tiene que conocer el funcionamiento de niveles inferiores.

Dado que los procesos no pueden acceder directamente al directorio de otros procesos mientras que el *Kernel* sí puede, tanto los clientes como los servicios utilizarán el *Binder Driver* para comunicarse.

La forma de interactuar con el Binder es utilizando su biblioteca de espacio de usuario, *libbinder.so*. Esta biblioteca implementa la mayoría de las operaciones de bajo nivel y estructuras de datos (por ejemplo, *Parcel*) utilizadas en la comunicación a nivel nativo, es el punto de entrada a la funcionalidad del Binder en el espacio de usuario.

Los clientes realizan peticiones a los servicios a través de *Binder Transactions*, que contienen un *Binder Token*, el código que identifica el método para ejecutar, una carga de datos útil, y el PID y UID del proceso que ha iniciado la transacción, el cual es añadido por el *Binder Driver*. [2]

Un cliente de alto nivel sólo se preocupa por el uso de un servicio arbitrario registrado en el sistema, del que puede obtener la referencia mediante el *ServiceManager* (Binder *CONTEXT_MGR*). No es necesario que clientes y servicios, implementen el *Binder Protocol* o invoquen la funcionalidad que ofrece *libbinder.so* de forma directa, pueden delegar en un *Proxy* en la parte cliente, y un *Stub* en la parte servidor.

La petición que realiza un cliente a un servicio invocará un conjunto de funcionalidades en la interfaz del Proxy, entre las que se encuentran la obtención de la referencia al servicio a través de una *Binder Interface* y el *marshalling* de los datos a adjuntar en la petición.

El punto de acceso al *Binder Driver* en espacio de usuario es a través del fichero */dev/binder*. Este ofrece una API simple basada en las llamadas al sistema *open, reléase, poll, mmap, flush,* y *ioctl*. Para invocar una transacción, se debe realizar una llamada *ioctl* donde se incluyan en sus argumentos el descriptor de fichero de */dev/binder*, el código de transacción y la estructura de la transacción.

Cuando se invoca, el *Binder Driver* resuelve el *Binder Object* destino, copia los datos en el espacio de direcciones del servidor y despierta un hilo en el proceso de servidor para procesar la petición. El servidor sigue los mismos pasos para enviar la respuesta al cliente, con la diferencia de que no debe lanzar un nuevo hilo, pues el cliente estará bloqueado esperando la respuesta. [7]

El *Stub* del servidor escucha continuamente las peticiones entrantes. Cuando se activa por una petición, resuelve el código de operación que identifica el método que se solicita del servicio, y hace el *unmarshalling* de los argumentos con los que invoca al método requerido. Por último, manda una respuesta al cliente con el valor de retorno del método.

Es posible añadir otra capa de abstracción para que los clientes no se tengan que preocupar de que están utilizando el *Binder Driver*, interfaces *Proxy* ni ningún mecanismo relacionado con IPC. Se puede recurrir a *managers* de alto nivel para abstraer toda esta complejidad. Es habitual que los servicios del sistema solo expongan un conjunto de APIs donde ofrecen su funcionalidad a los clientes a través de los *managers*, estas son las AIDLs. [6]

2.6.3. Transacciones

Cada paquete enviado entre procesos se denomina *Binder Transaction*. Toda transacción contiene un *Binder Token* que identifica el *Binder Object* de destino, el código de la acción que se solicita, un *buffer* de datos en bruto y el identificador del proceso que realiza la petición (PID/UID).

Cuando el código de petición enviado al invocar la llamada al sistema *ioctl* sobre el *Binder Driver* coincide con *BINDER_WRITE_READ*, entonces se trata una petición de IPC. Este código de petición es la base de las operaciones de IPC sobre el *Binder Driver*. Cuando aparece este código, la estructura del *buffer* de datos a la que apunta el tercer argumento de la invocación de *ioctl* es una estructura de tipo *struct binder_write_read,* mostrada en la Figura 3, y se puede encontrar en *binder.h*.

```
struct binder_write_read {
    signed long write_size; /* bytes to write */
    signed long write_consumed; /* bytes consumed by driver (TRANSACTION) */
    unsigned long write_buffer;
    signed long read_size; /* bytes to read */
    signed long read_consumed; /* bytes consumed by driver (REPLY) */
    unsigned long read_buffer;
} ;
```

Figura 3: Estructura binder_write_read

El modelo de IPC que ofrece el Binder está basado en un mecanismo de petición-respuesta. Esto implica que un proceso que hace una petición se va a quedar bloqueado hasta que llegue la respuesta, a no ser que se indique específicamente que es una petición sin respuesta mediante el *flag TF_ONE_WAY*. En general, una transacción implica dos mensajes, el de petición y el de respuesta. La comunicación unidireccional está limitada a la gestión de características internas, como el *death notification*. [2]

Las estructuras de la petición y la respuesta estarán contenidas en el *struct binder_write_read* tras la invocación de la llamada al sistema *ioctl*. Los datos de la petición y la respuesta se encuentran en las direcciones a las que apuntan *write_buffer* y *read_buffer* respectivamente. Ambos *buffers* tienen un prefijo de 4 *bytes* que identifica el tipo de petición y de respuestas. Todos los códigos posibles se encuentran en *enum BinderDriverCommandProtocol*.

Si el prefijo del *write_buffer* coincide con el código *BC_TRANSACTION*, se tratará de una transacción de IPC. En este caso, el contenido de *write_buffer* va a tener una estructura de tipo *struct binder_transaction_data*, la cual también está declarada en *binder.h* (ver Figura 4).

```c
struct binder_transaction_data {
    union {
     size_t handle;
     void *ptr;
    } target;
    void *cookie;
    unsigned int code;
    unsigned int flags;
    pid_t sender_pid;
    uid_t sender_euid;
    size_t data_size;
    size_t offsets_size;
    union {
     struct {
       const void *buffer;
       const void *offsets;
    }ptr;
     uint8_t buf[8];
    }data;
};
```

Figura 4: Estructura binder_transaction_data

La estructura de la transacción contiene:

- *target*: identifica el *Binder Object* destino. Contempla la diferencia entre la invocación de un objeto en el espacio de memoria de otro proceso o del propio proceso.
- *data.ptr.buffer:* dirección de memoria que apunta a un *buffer* "serializado", donde cada entrada se corresponde con la identificación de un comando y los argumentos con los que se invoca. Este comando normalmente identifica la AIDL de un servicio del sistema (por ejemplo, *MediaPlayer*), y los argumentos son los que se le pasan al método de la AIDL al invocarse. Se corresponde con un objeto de tipo *Parcel* en alto nivel.
- *code:* identifica al método invocado. El código se resuelve por el *Stub* del servicio, el cual hace el *unmarshalling* de los comandos y argumentos contenidos en el *Parcel* que acompaña a la petición e invoca al correspondiente método de la AIDL. Estos métodos están ordenados de forma que su posición de declaración en el fichero coincide con su código identificativo, empezando por el valor *IBinder::FIRST_CALL_TRANSACTION*, definido con el valor 1.

Esta estructura proporciona una información muy significativa sobre las invocaciones entre procesos y el paso de datos, el cual se hace en claro a no ser que se realice un cifrado de alto nivel. La Figura 5, obtenida del artículo *Man in the Binder* [7] muestra de forma ilustrativa esta jerarquía de estructuras.

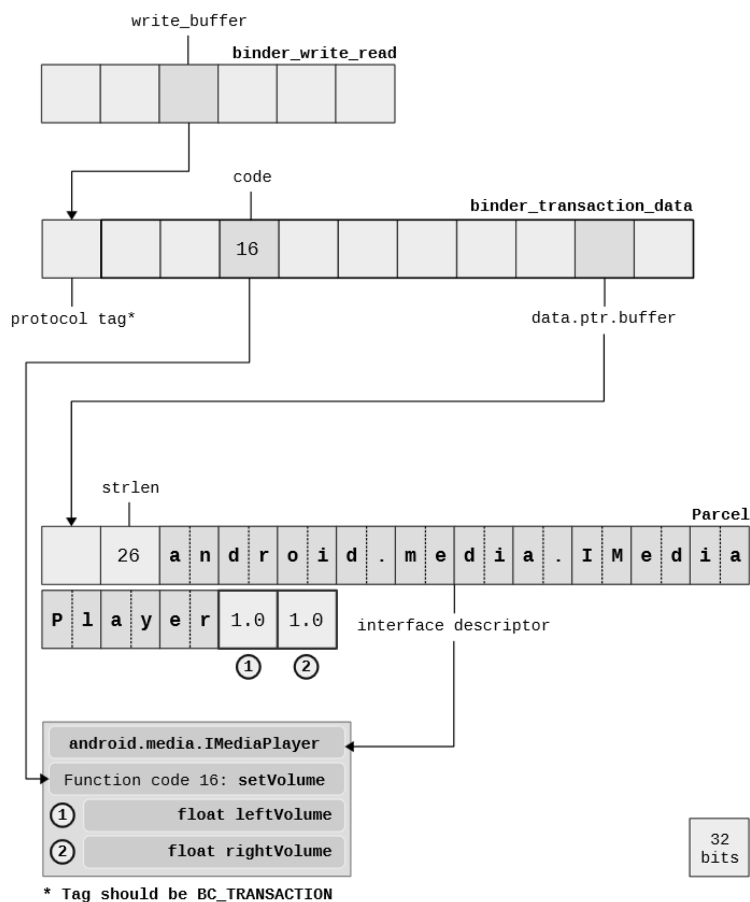


Figura 5: Estructura de datos en el binder

2.6.4. Android Interface Definition Language (AIDL)

La AIDL establece un protocolo de que relaciona la petición de un cliente con el método invocado en un servicio cuando se utiliza Binder IPC. Esta define una interfaz donde se declaran los métodos que ofrece un servicio remoto.

La AIDL proporciona una clase Java que utiliza la interfaz definida y se puede utilizar tanto como *Proxy*, para enviar peticiones a un servicio, así como *Stub* del servicio que escucha de forma continua por las peticiones entrantes. La AIDL se comparte entre cliente y servicio.

No es obligatorio que un servicio implemente su AIDL. El cliente y el servicio pueden implementar su *Proxy*, *Stub* e interfaz común de los métodos que ofrece el servicio de forma manual a nivel nativo.

2.7. Memoria Compartida en Android

El espacio de memoria física al que un proceso puede acceder está limitada. Un proceso solo puede acceder a direcciones de memoria físicas mapeadas en su espacio de memoria virtual.

La memoria virtual es un mecanismo de administración de la memoria física implementado mediante la combinación de técnicas software y hardware. Realiza el mapeo de las direcciones de memoria visibles para un proceso, llamadas direcciones virtuales, en las direcciones de memoria físicas.

Los principales beneficios de la memoria compartida son la liberación de un proceso de tener que gestionar su propio espacio de direcciones, el incremento de seguridad por el aislamiento de memoria, y la posibilidad de utilizar, de forma conceptual, más memoria de la disponible físicamente, utilizando la técnica de paginación.

La región de memoria compartida consiste en un segmento de memoria física que puede ser mapeada por dos o más procesos con el fin de establecer una comunicación entre ellos o para evitar hacer copias redundantes de datos (por ejemplo, las bibliotecas dinámicas). Existe una zona en la memoria física con este fin. El mapeo de memoria depende de la gestión del *Kernel* y de las políticas de seguridad que establezca.

Una aproximación de la estructura lógica de la memoria física tal y como se trabaja por el *Kernel* de Linux se muestra en la Figura 6:

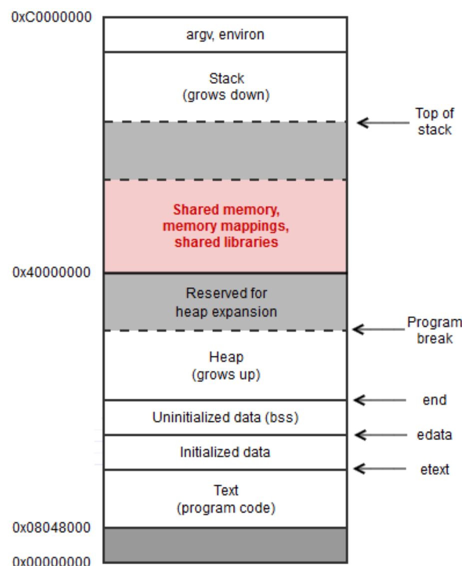


Figura 6: Estructura de la memoria física en el Kernel de Linux

La comunicación en memoria compartida el método más rápido de IPC disponible, pues, una vez que ha sido mapeada en el espacio de direcciones virtuales de los procesos involucrados en la comunicación, la interacción directa con el *Kernel* deja de ser necesaria para enviar datos, es decir, no es necesario invocar llamadas al sistema para comunicarse como ocurre con otros tipos

de mecanismos de IPC (por ejemplo, tuberías, colas de mensajes, etc.), sin contar con la gestión que este haga en segundo plano [8].

Se requiere que los procesos establezcan su propio mecanismo de sincronización entre lecturas y escrituras de memoria.

2.7.1. Address space layout randomization (ASLR)

Es una técnica de seguridad informática por la que se organizan de forma aleatoria las posiciones del espacio de direcciones de áreas de datos clave de un proceso, incluyendo la base del ejecutable y las posiciones de la pila, el *heap* y las bibliotecas.

Lo único que se conoce es que un determinado símbolo dentro de un objeto compartido va a tener el mismo desplazamiento en memoria desde la dirección virtual base del objeto cargado en el proceso local, la cual se puede conocer, pues se encuentra en /proc/<PID>/maps.

Conociendo la dirección virtual base de un objeto compartido en dos procesos, y la dirección virtual de un símbolo (por ejemplo, una función) de ese objeto compartido en uno de los procesos se puede calcular la dirección virtual del símbolo en el otro proceso. [9]

remote_address = remote_base + (local_addrees – local_base)

2.7.2. Android Shared Memory (ashmem)

Como Android está basado en el *Kernel* de Linux, también habilita la compartición de memoria entre procesos, pero utiliza su propio sistema de asignación de memoria compartida, *ashmem*.

Android Shared Memory (*ashmem*) es un componente del sistema operativo Android que facilita la compartición y conservación de memoria compartida. Se gestiona a través de un controlador a nivel de *Kernel*. Está basado en la técnica de compartición de fichero en memoria en la que una región de un fichero es proyectada directamente en memoria, de modo que los cambios en memoria se propagan automáticamente al fichero.

Un proceso puede crear una zona de memoria en *ashmem* siguiendo estos pasos:

- Abre el fichero del sistema */dev/ashmem* obteniendo un descriptor de fichero.
- Invoca la llamada al sistema *ioctl* con código de operación *ASHMEM_SET_NAME* sobre el descriptor de fichero para dar un nombre a la región. El fichero creado se va a llamar */dev/ashmem/<NAME>*.
- Invoca la llamada al sistema *ioctl* con código de operación *ASHMEM_SET_SIZE* sobre el descriptor de fichero para dar un tamaño fijo en *bytes* al fichero.

El fichero creado en *ashmem* puede ser utilizado por diferentes procesos, pero el descriptor de fichero que se obtiene al abrirlo o el puntero a memoria compartida resultante de mapearlo con la llamada al sistema *mmap* es dependiente del proceso.

La llamada al sistema de mapeo tiene que ser invocada sobre el mismo descriptor de fichero por todos los procesos que quieran compartir ese fichero en memoria. El mapeo se hace visible en */proc/<PID>/maps.*

2.8. Análisis Dinámico de Binder IPC

El análisis de malware y explotación de sistemas en ordenadores convencionales ha sido frecuentemente estudiado en la literatura [10] [1] [11] [12] [13]. Es ampliamente aceptado que el análisis dinámico, es decir, el análisis de un proceso en ejecución, es indispensable. El malware se incluye a menudo en códigos ofuscados impracticables en un análisis estático. Además, el rastro que deja el malware tras ser ejecutado en un sistema a menudo no proporciona una cantidad de datos significativa para conocer su comportamiento, es necesario recopilar información durante la ejecución.

Se han utilizado muchas técnicas de análisis en tanto en entornos software y hardware virtualizados o emulados como en entornos reales para llevar a cabo la introspecciones de la ejecución de actividades maliciosas. [13]

A continuación se van a comentar dos trabajos que hoy en día son estado del arte en sistemas de detección de comportamientos malware. Uno en un entorno emulado y otro en un entorno real.

2.8.1. Análisis en Entorno Android Emulado – CopperDroid

CopperDroid [1] es un sistema de análisis dinámico para la reconstrucción de comportamientos de malware. Está basado en *Virtual Machine Introspection* (VMI), la cual consiste en una máquina virtual anfitrión que permite monitorizar la interacción con el *Kernel* mediante la interceptación de las llamadas al sistema.

La filosofía que siguen los diseñadores de esta herramienta es que, para conseguir una reconstrucción de comportamiento completa, hay que combinar la monitorización de IPC con la monitorización de las rutinas del sistema operativo, ya que estas últimas pueden ser las precedentes o consecuentes de IPC.

2.8.1.1. *Arquitectura e Interceptación*

La herramienta consiste en un sistema operativo Android, ejecutado en un emulador Android personalizado, llamado *CopperDroid Emulator*, que a su vez está soportado por una arquitectura hardware emulada con QEMU.

El objetivo del sistema es generar perfiles de comportamiento a partir del reconocimiento de ciertos patrones de interacciones a bajo nivel.

Para ello, el sistema intercepta las llamadas al sistema involucradas de forma directa en IPC y RPC. Dado que Android IPC está basado principalmente por el *Android Binder*, el sistema captura las *Binder Transactions* gestionadas por el *Binder Driver* cuando se invoca la llamada al sistema *ioctl* sobre él. Con los datos interceptados, realiza el *unmarshalling* de los objetos *Parcel* incluidos en las transacciones, para recuperar la información sobre las comunicaciones remotas.

Copperdroid También enriquece la recolección de datos capturando las llamadas al sistema regulares, a partir de las que es posible construir grafos de dependencia atendiendo a subconjuntos de invocaciones. El rastreo de las llamadas al sistema está basado en la arquitectura ARM emulada por QEMU, el cual está programado para interceptar la interrupción software que genera una llamada al sistema cuando se ejecuta la instrucción *swi*.

2.8.1.2. *Análisis de Comportamientos*

Es importante hacer notar que esta tarea de *unmarshalling* es muy costosa y compleja, ya que requiere conocer en profundidad todas las posibles interfaces de Android que van a ser invocadas a través del Binder. El algoritmo de *unmarshalling* que utilizan es capaz de distinguir entre tipos de datos primitivos o básicos, objetos de clases de Android y referencias a otros *Binder Objects*. Este algoritmo se muestra en la Figura 7.

En caso de ser tipos de datos primitivos, utiliza las funciones de *unmarshalling* que proporciona el la clase *Parcel* (por ejemplo, *readInt()*). En caso de que sean objetos, el algoritmo aplica "*Java reflection*". Consiste en obtener una referencia al campo *CREATOR* que contienen los objetos que implementan la clase *Parcelable*. Con esta referencia es posible invocar el método *createFromParcel()* para hacer el unmarshalling del objeto de alto nivel. Si el objeto incluido en el *Parcel* no es ni un tipo básico ni un objeto de clase, el algoritmo comprueba los primeros 4 *bytes* del objeto en busca de alguna referencia a la interfaz *IBinder*.

```
Data: Marshalled binder transaction and data types
      (determined through the AIDL)
Result: Unmarshalled binder transactions
1  while data → marshalled do
2      determine type of marshalled item;
3      if type → primitive then
4          automatically apply correct parcelable read/create
           functions;
5          append string repr. to results;
6      else
7          locate CREATOR field for reflection;
8          use java reflection to get class object;
9          for every class field do
10             if field → primitive then
11                 append string repr. to results;
12             else
13                 explore field recursively;
14                 append string repr. to results;
15             end
16         end
17         if CREATOR → not found and buffer → binder
           reference type then
18             Unmarshall Binder reference
19         end
20     end
21 end
```

Figura 7: Algoritmo de *unmarshalling* usado en *Copperdroid*

La construcción de grafos de dependencia sobre las llamadas al sistema observadas combinado con la información extraída de IPC permite reconstruir perfiles de comportamiento de las aplicaciones.

2.8.2. Análisis en Entorno Android Real

Una investigación realizada en 2012 por investigadores de la universidad de Wuhan y publicada a través Springer-Verlag propone una herramienta de monitorización de IPC mediante la interceptación de llamadas al sistema en espacio de usuario de un dispositivo Android real. [12]

2.8.2.1. Arquitectura e Interceptación

Como se ha mencionado en secciones anteriores, el sistema Android ofrece una serie de utilidades básicas, tales como el envío y la recepción de SMS, los gestores de multimedia, etc. Los componentes que ofrecen esta funcionalidad son servicios del sistema, los cuales residen en procesos servidor lanzados por el sistema. Una aplicación cliente accede a estos servicios a través de su AIDL y cuando los consume genera actividad IPC.

A pesar de la existencia de una gran cantidad de servicios del sistema registrados, esta investigación sugiere que solo existen tres procesos servidor encargados de la administración de estos servicios.

- *com.android-phone*: servicios relacionados con las telecomunicaciones (por ejemplo, SMS, llamadas telefónicas, etc.).
- *mediaserver*: servicios relacionados con multimedia (por ejemplo, grabación de video y voz).

- *system_server*: otros servicios (por ejemplo, como el de geolocalización, conexiones de red, etc.).

Es por tanto que los autores de esta investigación proponen monitorizar las transacciones de Binder IPC que lleguen a los procesos servidores. El mecanismo de petición-respuesta les permite obtener la misma transacción en la parte cliente que en la parte servidor, y consideran más eficiente y estable obtener datos de todos los clientes a la vez.

El sistema está compuesto por un módulo que se encarga de inyectar una biblioteca dinámica en el proceso objetivo (ver Figura 8). Esta biblioteca redirecciona la llamada al sistema *ioctl* que el proceso tiene mapeada de forma original por una rutina implementada en la biblioteca donde se extrae la transacción. Este redireccionamiento se hace a través de un "*patching"* de la PLT. Cada *Binder Transaction* interceptada se extrae a una aplicación utilizando un *socket* local, donde se almacena y evalúa calculando el nivel de amenaza (*Threat Point*) que supone.

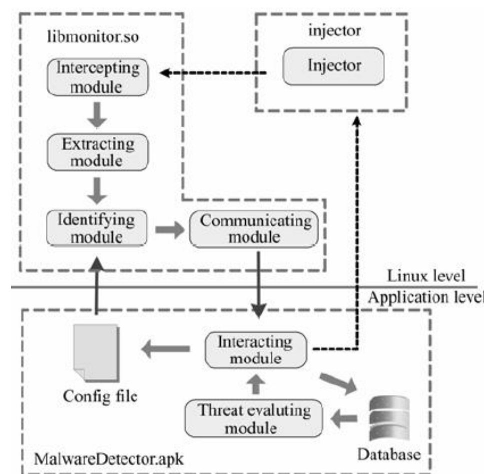


Figura 8: Esquema de monitorización propuesto en [12]

2.8.2.2. *Análisis de Comportamientos*

El sistema utiliza un algoritmo de pesos para evaluar el nivel de amenaza de las aplicaciones. Las estadísticas extraídas del análisis de aplicaciones que no contienen malware consiguen clasificar una serie de patrones de comportamiento.

Las combinaciones de estos patrones usadas de forma habitual por las aplicaciones, como el acceso a Internet, son evaluados con menor peso que los patrones poco frecuentes, como la iniciación de una llamada telefónica. Además, los distintos grupos de permisos de Android se pueden clasificar en diferentes niveles de peligrosidad en consonancia con los patrones.

Las piezas de malware a menudo suelen presentarse como patrones que se repiten cada cierto intervalo de tiempo. También hay que prestar especial atención a las aplicaciones que no tienen

asociadas interfaces de usuario. Siempre que se dan estos casos el nivel de amenaza de la aplicación en la que se notifican se incrementa considerablemente.

2.9. Detección de Entornos de Análisis

Una cuestión que no se tiene en cuenta habitualmente a la hora de desarrollar sistemas de análisis dinámico en Android es la capacidad que tiene el malware de detectar entornos de análisis, tales como emuladores o máquinas virtuales.

Una pieza de malware puede detectar entornos de análisis y rechazar la ejecución de las actividades maliciosas.

La estrategia que se usa más habitualmente para este propósito es la inspección del sistema donde se está ejecutando. La idea consiste en comprobar características muy específicas, como algunas claves en registros, procesos en segundo plano, interceptación de funciones o la aparición de direcciones IP en las conexiones de red que utilizan algunas herramientas de análisis.

Otro enfoque se basa en aprovecharse de que la mayoría de sistemas de análisis utilizan emuladores o máquinas virtuales como plataforma de ejecución de malware. La presencia de estas plataformas se puede detectar comprobando las discrepancias con respecto a un sistema operativo Android instalado en hardware real [10]. Las discrepancias pueden ser estáticas, dinámicas y condicionadas por el monitor de máquina virtual (hipervisor).

Las diferencias estáticas permiten diferencias emuladores de dispositivos reales de forma inmediata. Se pueden comprobar propiedades del sistema tales como el código IMEI del dispositivo, el código IMSI de la tarjeta SIM, el número de núcleos del dispositivo, pues en los emuladores suele ser uno, los periféricos disponibles, ya que un emulador no suele soportar puerto de tipo USB On-The-Go, y la configuración del hardware de red. Estas discrepancias se pueden camuflar mediante el desarrollo de emuladores más completos.

Las diferencias dinámicas, en cambio, no pueden ser disimuladas fácilmente. En un emulador no se suele contemplar la funcionalidad plena de todas las interfaces. Una aplicación puede comprobar si es posible recibir SMS de la red móvil. Además, los dispositivos móviles contienen una gran cantidad de sensores entre los que se pueden encontrar GPS, cámara, GSM, WiFi, Bluetooth, RFC, termómetros internos, voltímetros, etc. Los emuladores no suelen dar soporte a la simulación de la reacción de un sensor ante un estímulo físico, los cuales pueden ser influenciados por una aplicación. Un incremento de uso de la CPU, por ejemplo, aumenta la temperatura interna, el voltaje de los núcleos y el consumo de batería.

Por último, la ejecución sobre un monitor de máquina virtual puede ser detectada. QEMU es detectable por sus políticas de *caching* y planificación. [11]

Un sistema de análisis que es indetectable es el que utiliza un sistema operativo original, sin modificaciones, el cual se ejecuta sobre un dispositivo real. Comúnmente se denominan *bare-metal*.

El reto que se plantea para este tipo de sistemas basados en *bare-metal* es la extracción de los datos generados. Ninguna extracción de datos es posible sin introducir algún tipo de componente involucrado en el análisis, el cual incumple el principio de transparencia y hace al sistema detectable. [10]

La herramienta *BareDroid* [11] propone un sistema que puede ser usado para realizar análisis a gran escala sobre aplicaciones Android que se ejecutan en dispositivos reales. El objetivo del trabajo es evitar, en la medida de lo posible, la detección del entorno de análisis. Para ello, proponen un mecanismo de restauración del estado inicial del dispositivo móvil, el cual no se encuentra comprometido. Esta herramienta consigue restaurar cada partición del dispositivo antes de un análisis. De esta forma, el malware encontrará un sistema original y sin modificaciones.

2.10. Interceptación de Llamadas al Sistema en Entornos Android Reales

El planteamiento más común para describir el comportamiento del malware en Android consiste en interceptar las llamadas al sistema que invoca el proceso sobre el que se ejecuta el malware, y extraer los datos obtenidos para realizar un análisis de comportamiento fuera del dispositivo.

Una de las consideraciones de mayor importancia durante la monitorización de llamadas al sistema es la transparencia con la que actúa el sistema de análisis. Se han utilizado diferentes técnicas con este fin.

A continuación se van a comentar tres posibles alternativas para interceptar las llamadas al sistema invocadas por un proceso.

2.10.1. Hook de Llamadas al Sistema

Esta técnica consiste en la redirección de una llamada al sistema ejecutada por un proceso para que en lugar de invocar a la función original, invoque una conocida, donde se trate de una forma u otra dependiendo del objetivo.

Un ejemplo se muestra en la Figura 9. Sea un programa escrito en lenguaje C que se llama test (test.c), y dos bibliotecas dinámicas (libtest1.so y libtest2.so) que implementan las funciones *libtest1()* y *libtest2()* respectivamente, las cuales invoca test. Estas dos funciones, a su vez,

recurren a la llamada al sistema *puts*, implementada en la biblioteca dinámica estándar *libc.so* (*bionic* en Android).

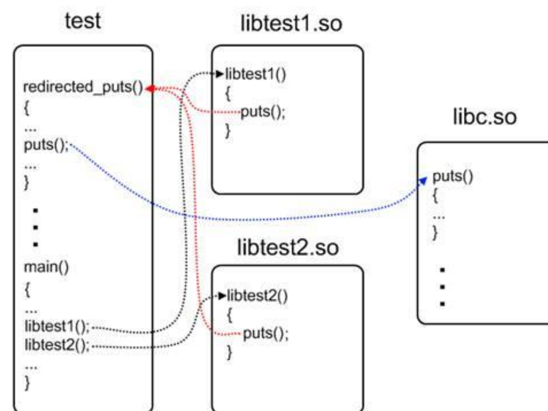


Figura 9: Hooking de llamadas al sistema [14]

La redirección consiste en reemplazar la dirección de la función *puts()* que ambas librerías tienen mapeada por un dirección a una función de creación propia (*redirected_puts()*) que acepte la misma estructura de paso de parámetros que la función original. Dentro de esta función propia, además de las alteraciones que se pretendan realizar, es necesario invocar a la función original para poder retornar el valor resultante a la función que ha invocado a *puts()* y así evitar bloqueos o alteraciones en el flujo del proceso.

La redirección se suele hacer desde una biblioteca dinámica que tiene que ser cargada por el proceso objetivo.

Una biblioteca dinámica o compartida es una biblioteca que se carga y se enlaza en tiempo de ejecución, no de compilación. Estas bibliotecas se mapean en el espacio de memoria virtual de cada proceso permitiendo que mantener privados datos estáticos y globales para cada proceso. La sección de memoria física donde reside una biblioteca compartida es en la zona de memoria compartida.

Esta técnica está basada en el planteamiento de que no todas las bibliotecas dinámicas son auto-contenidas, es decir, confían en otras bibliotecas, las cuales pueden ser directamente las bibliotecas estándar (*libc.so* y *libstdc++.so*), donde están implementadas las rutinas de las llamadas al sistema en espacio de usuario.

Cuando una biblioteca dinámica es cargada por un proceso se ejecuta su constructor, que es donde suele aplicarse la redirección. Un método muy utilizado es el *patching* o parcheo de la *Procedure Linkage Table (PLT)*.

La mayoría de las bibliotecas dinámicas usadas en distribuciones basadas en el *Kernel* de Linux tienen formato ELF (*Executable and Linkable Format*). Por lo que cuentan con una *Procedure Linkage Table* (PLT) y una *Global Offset Table* (GOT).

La PLT es un *array* de manejadores, uno por función mapeada de algún módulo externo. Se utiliza para invocar procedimientos y funciones externas cuyas direcciones son desconocidas en tiempo de enlazado. Serán resueltas por el enlazador dinámico (*dynamic linker*) en tiempo de ejecución. GOT es otro *array* que se usa de forma similar a la PLT.

Las llamadas de las funciones que una biblioteca importa se resuelven a través de la PLT y la GOT. La redirección se va a hacer una única vez por función y se va a aplicar a la primera instrucción de un elemento específico en la PLT.

Para poder hacer la redirección hay que conocer:

- El nombre de enlazado de la biblioteca objetivo.
- La dirección virtual en la que está cargada.
- El nombre o símbolo de la función a ser suplantada.
- La dirección de la función sustituta.

2.10.2. Registro de SELinux

Esta técnica permite obtener un informe completo de las operaciones ejecutadas por una aplicación sin introducir modificaciones en el núcleo o añadir componentes específicos.

SELinux es el módulo de seguridad *Security-Enhanced Linux* (SELinux) para el *Kernel* de Linux. SELinux hace cumplir una política de seguridad que se extiende sobre todos los procesos, los objetos y las operaciones basadas en etiquetas de seguridad que pueden trabajar con información de seguridad de procesos y objetos. Como mecanismo de Control de Acceso Obligatorio (MAC), SELinux es capaz de aislar aplicaciones defectuosas y maliciosas, incluso las que se ejecutan en modo superusuario.

Entre otras funciones, SELinux puede controlar todas las posibles interacciones entre aplicaciones de Android a nivel de núcleo, también puede controlar todo el acceso recursos del sistema por parte de las aplicaciones.

En la práctica, antes de la ejecución de cada llamada al sistema, el *Kernel* consulta a SELinux para saber si un determinado proceso está autorizado para realizar la operación solicitada.

SELinux solo registra las operaciones denegadas por defecto, pero se puede modificar la política de SELinux para que registre otro tipo de información cuando es consultado por el Kernel [15].

El proyecto *BareDroid* utiliza esta técnica [11]. La modificación que se hace sobre la política de seguridad para obtener información de cada llamada al sistema consiste en la introducción de una regla de *auditallow* por cada regla *allow*. Es decir, por cada regla que permita la ejecución de una operación (*allow*), añade otra que registra el tipo de operación (*auditallow*) [16], con el fin de obtener un informe completo de las operaciones realizadas por una aplicación. A partir de este informe, se puede extraer información de las operaciones sobre fichero. Es utilizado

como una medida empírica de la cantidad de comportamientos realizados por la aplicación maliciosa.

2.10.3. Strace

La herramienta *strace,* incluida en muchas distribuciones Linux, permite trazar llamadas al sistema y señales para un determinado proceso [17]. Esta herramienta permite depurar un proceso mediante la monitorización de llamadas al sistema que éste hace, así como las señales que recibe. Permite mostrar tanto el nombre como los argumentos del mismo, y además realiza simples estadísticas como la frecuencia de cada llamada, el tiempo de ejecución de las mismas, etc.

El problema de usar *strace* en un sistema Android es que no viene incluido en el *Kernel* de fábrica en las actuales versiones de Android, y por lo tanto es necesario compilarlo e incluirlo. Además, esta herramienta muestra los datos en local (es decir, en el dispositivo), y no permite mandar la información a un servidor remoto, por lo que se ha descartado su uso en este Trabajo de Fin de Grado.

Capítulo 3: Análisis

3.1. Planteamiento del Problema

El objetivo de este Trabajo de Fin de Grado es construir una herramienta que permita monitorizar los eventos que conformen el comportamiento de una aplicación en Android para poder extraer posibles patrones que permitan detectar actividades maliciosas.

Los sistemas de detección de comportamientos de malware que se construyen sobre emuladores pueden ser detectados por las propias muestras de malware comprobando una serie de características que revelan discrepancias entre una plataforma emulada y un sistema operativo sin modificar instalado sobre un dispositivo real (*bare-metal*).

El reto al que se enfrenta un sistema de análisis de malware basado en un *bare-metal* consiste en la extracción de los datos generados del dispositivo móvil. La extracción es impracticable sin introducir algún componente de análisis en el sistema anfitrión, por lo que se viola el requisito de transparencia haciendo al sistema de análisis detectable.

El sistema que se propone tiene que minimizar la probabilidad de ser detectado. El componente interno en el dispositivo debe limitarse a la interceptación y extracción de los datos. La interceptación se considera una tarea crítica donde el rendimiento y la transparencia son cruciales, mientras que la extracción de los datos puede ser delegada a otro proceso. La comunicación entre procesos debe ser lo más ligera posible, evitando siempre que sea posible la interacción directa del *Kernel*.

3.2. Descripción de la solución

Para poder recuperar los datos que definen el comportamiento de un proceso determinado en Android se van a interceptar las llamadas al sistema que invoca dicho proceso. Se presta especial atención a las llamadas al sistema relacionadas con comunicación a través del *Binder Kernel Driver*, mecanismo de IPC fundamental en Android. El interés recae tanto sobre las llamadas que invocan servicios remotos que residen en otros procesos de forma directa (*ioctl*), como las llamadas que pueden enriquecer el análisis de comportamientos a bajo nivel (*write, read, connect…*).

El sistema de monitorización de las llamadas al sistema se va a construir para un dispositivo móvil sobre el que se ejecuta un sistema operativo Android, es decir, un entorno real, no simulado. Esto va a garantizar precisión y completitud durante la recolección de datos.

Se deberá tener en cuenta en todo momento la portabilidad del sistema y su eficiencia en el uso de los recursos reducidos con los que cuenta un dispositivo móvil. Para ello, la monitorización se realizará en espacio de usuario, sin necesidad de modificar el *Kernel* del sistema operativo, ya que en caso de querer desplegarlo en distintos dispositivos sean necesarios los mínimos cambios posibles. El único requisito es que el sistema operativo permita la escalada de privilegios a superusuario y que la política de seguridad de SELinux adquiera el nivel más permisivo.

La tarea de interceptación la va a llevar a cabo un proceso a nivel nativo, que se encarga de interceptar las llamadas al sistema más significativas en el análisis de comportamientos en Android. Por motivos técnicos de transparencia, funcionalidad y rendimiento, la extracción la va a realizar otro proceso parte del sistema, y que va a enviar los datos a través de la red a un servidor de análisis, donde se van a interpretar y almacenar.

3.3. Casos de Uso

Se han identificado dos casos de uso para este sistema. Los casos de uso se van a representar mediante diagramas de secuencia.

3.3.1. Caso de uso CU-01: Interceptación de Binder Transactions.

Este caso de uso representa la interceptación del inicio de una transacción con el Binder.

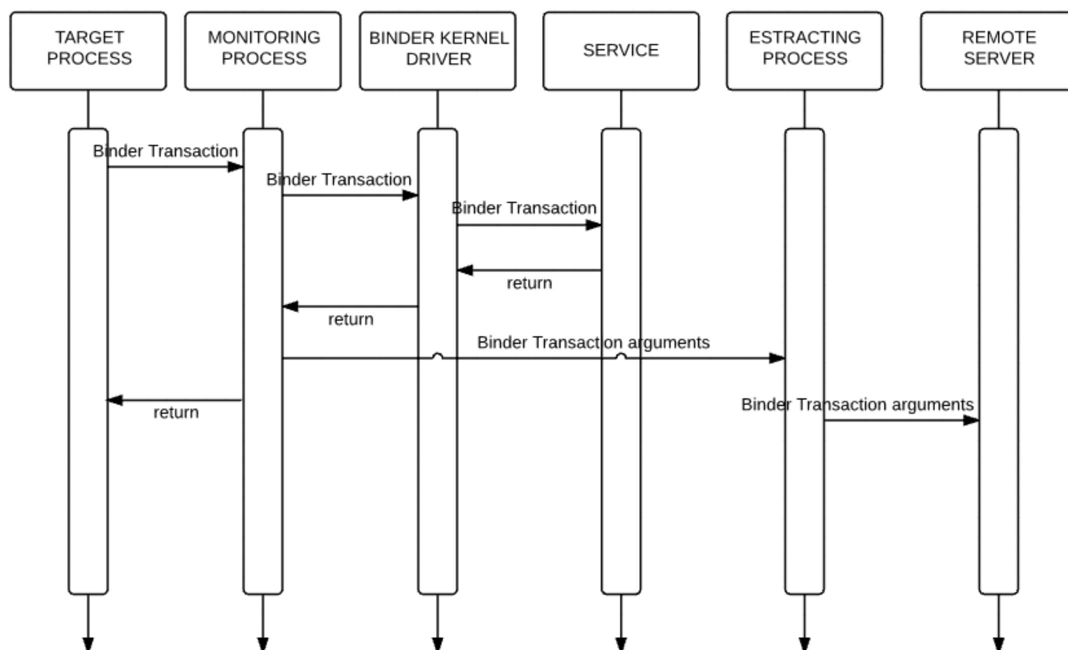


Figura 10: CU-01

3.3.2. Caso de uso CU-02: Interceptación de llamadas al sistema regulares.

Este caso de uso representa la interceptación de una llamada al sistema regular.

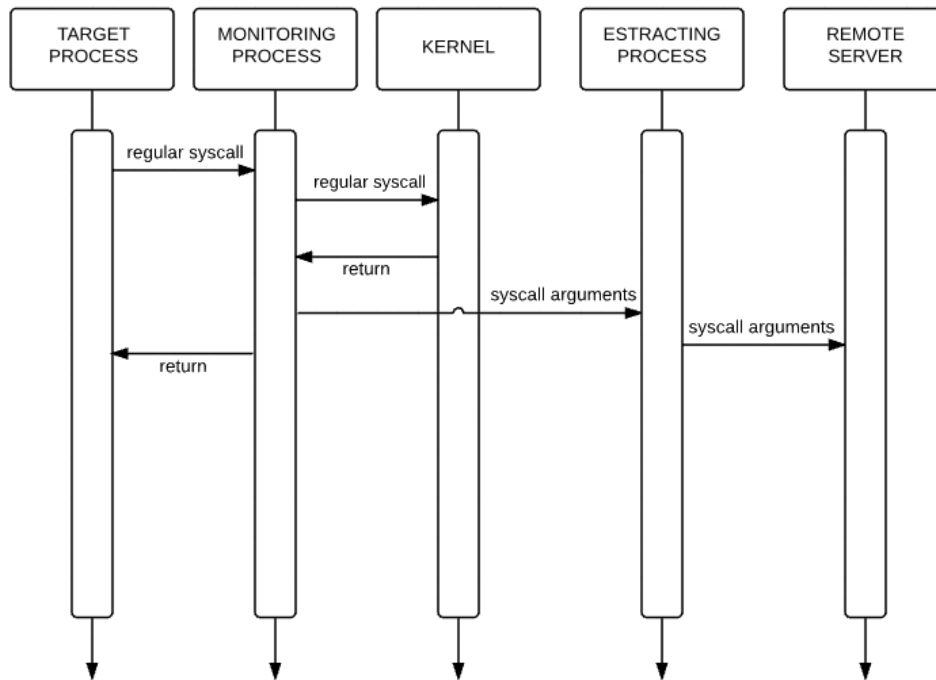


Figura 11: CU-02

3.4. Especificación de Requisitos.

En la especificación de requisitos se pone por escrito la naturaleza exacta de la aplicación. El nivel de detalle que expone debe ser completo, pero no redundante.

3.4.1. Plantilla de Requisito

Identificador: TIPO-XX	
Título:	
Proviene de:	
Prioridad:	☐ Alta ☐ Media ☐ Baja
Necesidad:	☐ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☐ Si ☐ No
Estabilidad:	☐ Estable ☐ Inestable
Descripción:	

Tabla 1: Plantilla de Requisito

Los campos recogidos en la anterior tabla tiene el siguiente significado:

- **Identificador:** permiten identificar unívocamente cada requisito.
- Título: título representativo del requisito.
- **Proviene de:** indica el origen del requisito, puede ser la entrevista con el cliente u otro requisito.
- **Prioridad:** especifica en qué fase del diseño técnico de la aplicación se debe incluir la funcionalidad de dicho requisito.
- **Necesidad:** indica el grado de importancia de dicho requisito en el proyecto.
- Verificabilidad: indica si la incorporación del requisito se puede evidenciar en el producto final.
- **Estabilidad:** indica la posibilidad de que un requisito pueda ser modificado de acuerdo con las impresiones del cliente sobre dicho requisito o con el diseño técnico.
- **Descripción:** se explica cuál es la funcionalidad o restricción que el requisito va a implicar en el proyecto.

3.4.2. Requisitos de Usuario.

Expresados en lenguaje comprensible por el cliente y de una forma no estructurada. Tienen poco nivel de detalle.

- **Requisitos de capacidad** (identificador CAP-XX): especifican las funciones y operaciones que el software debe realizar desde el punto de vista del cliente.
- **Requisitos de restricción** (identificador RES-XX) especifican la manera en el que el software debe resolver los problemas o alcanzar los objetivos. Incluyen limitaciones y las barreras que el software no puede traspasar.

3.4.2.1. Requisitos de Capacidad

Identificador: CAP-01	
Título:	Monitorizar aplicaciones Android.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema permitirá monitorizar información sobre la ejecución de aplicaciones en Android de forma dinámica.

Tabla 2: CAP-01

Identificador: CAP-02	
Título:	Interceptar IPC.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará los datos intercambiados en la invocación de servicios remotos de otros procesos por parte de la aplicación monitorizada.

Tabla 3: CAP-02

Identificador: CAP-03	
Título:	Interceptar actividad en fichero.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará los datos referentes a la apertura, cierre, escritura y lectura de ficheros por parte de la aplicación monitorizada.

Tabla 4: CAP-03

Identificador: CAP-04	
Título:	Interceptar actividad de red.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará los datos referentes a la conexión a interfaces de red, así como el envío y recepción de paquetes a través de estas interfaces por parte de la aplicación monitorizada.

Tabla 5: CAP-04

Identificador: CAP-05	
Título:	Ubicación temporal de eventos.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	Los paquetes de datos extraídos serán marcados con una referencia temporal que permita su ordenación temporal.

Tabla 6: CAP-05

Identificador: CAP-06	
Título:	Envío a servidor externo.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	Los datos extraídos se enviarán a un servidor externo a través de la red.

Tabla 7: CAP-06

Identificador: CAP-07	
Título:	Persistencia de los datos.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	Los datos extraídos se almacenarán de forma persistente en el servidor externo.

Tabla 8: CAP-07

3.4.2.2. *Requisitos de Restricción*

Identificador: RES-08	
Título:	Entorno Android.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El entorno de monitorización será un dispositivo real con sistema operativo Android.

Tabla 9: RES-08

Identificador: RES-09	
Título:	No alterar el funcionamiento habitual.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	La interceptación de datos no alterará el funcionamiento habitual de la aplicación monitorizada.

Tabla 10: RES-09

Identificador: RES-10	
Título:	Interceptación transparente.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	La interceptación de datos debe ser transparente a la aplicación monitorizada y no modificar los eventos analizados.

Tabla 11: RES-10

Identificador: RES-11	
Título:	Extracción transparente.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El envío de los datos al servidor externo debe ser transparente a la aplicación monitorizada.

Tabla 12: RES-11

Identificador: RES-12	
Título:	Host del servidor externo.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☐ Esencial ☒ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El servidor externo residirá en un ordenador convencional.

Tabla 13: RES-12

Identificador: RES-13	
Título:	Integridad de los datos.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema debe garantizar que todos los datos interceptados lleguen al servidor garantizando su integridad y completitud.

Tabla 14: RES-13

Identificador: RES-14	
Título:	No cambiar el núcleo.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	La implementación del sistema no requerirá cambios en el núcleo del sistema operativo del dispositivo móvil.

Tabla 15: RES-14

Identificador: RES-15	
Título:	Acceso a la red habilitado.
Proviene de:	Cliente
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El dispositivo móvil contará con conexión de red.

Tabla 16: RES-15

Identificador: RES-16	
Título:	Estructura modular.
Proviene de:	Cliente
Prioridad:	☐ Alta ☒ Media ☐ Baja
Necesidad:	☐ Esencial ☒ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema tendrá una estructura modular que permita ampliar y mantener su funcionalidad en futuras versiones de Android y de la herramienta.

Tabla 17: RES-16

3.4.3. Requisitos de Sistema.

Expresados de forma técnica y precisa por el analista. Tienen una forma estructurada y alto nivel de detalle.

- **Requisitos funcionales** (identificador RF-XX): especifican la funcionalidad o servicios que el sistema debe proporcionar. Derivan de los requisitos de capacidad del usuario.
- **Requisitos negativos** (identificador RN-XX) especifican lo que el sistema no debe hacer. Se seleccionarán únicamente los que ayuden a aclarar los verdaderos requisitos. Derivan de los requisitos de capacidad y de restricción del usuario.
- **Requisitos no funcionales** (identificador RNF-XX) especifican restricciones y pautas en el sistema en desarrollo y en el proceso de desarrollo. Derivan de los requisitos de restricción del usuario.

Los argumentos de las llamadas al sistema se pueden encontrar en Linux Man Pages [18].

3.4.3.1. *Requisitos Funcionales*

Identificador: RF-01	
Título:	Interceptación dinámica.
Proviene de:	CAP-01
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará llamadas al sistema invocadas por los procesos monitorizados en tiempo de ejecución.

Tabla 18: RF-01

Identificador: RF-02	
Título:	Interceptar ioctl.
Proviene de:	CAP-02
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará la invocación de la llamada al sistema *ioctl* por parte del proceso monitorizado. En caso de la llamada inicie una *Binder Transaction* se extraerán los siguientes campos de su estructura: `target.handle`, `target.ptr`, `cookie`, `code`, `flags`, `sender_pid`, `sender_euid`, `data_size` y `offsets_size` además de los contenidos de los *buffers* `data.ptr.buffer` y `data.ptr.offsets`.

Tabla 19: RF-02

Identificador: RF-03	
Título:	Interceptar open.
Proviene de:	CAP-02, CAP-03
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará la invocación de la llamada al sistema *open* por parte del proceso monitorizado. Se extraerán los siguientes argumentos de la invocación: `pathname` y `flags`.

Tabla 20: RF-03

Identificador: RF-04	
Título:	Interceptar write.
Proviene de:	CAP-02, CAP-03
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará la invocación de la llamada *write* por parte del proceso monitorizado. Se extraerán los siguientes argumentos de la invocación: `fd, buf` y `count`.

Tabla 21: RF-04

Identificador: RF-05	
Título:	Interceptar read.
Proviene de:	CAP-02, CAP-03
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará la invocación de la llamada al sistema *read* por parte del proceso monitorizado. Se extraerán los siguientes argumentos de la invocación: `fd`, `buf` y `count`.

Tabla 22: RF-05

Identificador: RF-06	
Título:	Interceptar close.
Proviene de:	CAP-02, CAP-03
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará la invocación de la llamada al sistema *close* por parte del proceso monitorizado. Se extraerán los siguientes argumentos de la invocación: `fd`.

Tabla 23: RF-06

Identificador: RF-07	
Título:	Interceptar connect.
Proviene de:	CAP-04
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará la invocación de la llamada al sistema *connect* por parte del proceso monitorizado. Se extraerán los siguientes argumentos de la invocación: `socket`, `address`, `address_len`.

Tabla 24: RF-07

Identificador: RF-08	
Título:	Interceptar *send, sendto* y *sendmsg*.
Proviene de:	CAP-04
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará la invocación de las llamadas al sistema *send, sendto* y *sendmsg* por parte del proceso monitorizado. Se extraerán los siguientes argumentos de la invocación: `sockfd`, `buf`, `len`, `flags`, `dest_addr`, `addrlen` y `msg` (cada uno de su respectiva invocación).

Tabla 25: RF-08

Identificador: RF-09	
Título:	Interceptar *recv*, *recvfrom*, y *recvmsg*.
Proviene de:	CAP-04
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema interceptará la invocación de las llamadas al sistema *recv*, *recvfrom*, y *recvmsg* por parte del proceso monitorizado. Se extraerán los siguientes argumentos de la invocación: `sockfd`, `buf`, `len`, `flags`, `src_addr`, `addrlen` y `msg` (cada uno de su respectiva invocación).

Tabla 26: RF-09

Identificador: RF-10	
Título:	Referencia temporal de los paquetes.
Proviene de:	CAP-05
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	Los paquetes de datos extraídos serán marcados con una referencia temporal relativa que permita su ordenación temporal.

Tabla 27: RF-10

Identificador: RF-11	
Título:	Envío de paquetes a servidor externo.
Proviene de:	CAP-06
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema enviará los paquetes de datos interceptados a un servidor externo.

Tabla 28: RF-11

Identificador: RF-12	
Título:	Almacenamiento de paquetes.
Proviene de:	CAP-06
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El servidor almacenará la información en ficheros con formato *.csv* en el sistema de ficheros del disco duro.

Tabla 29: RF-12

3.4.3.2. Requisitos Negativos

Identificador: RN-01	
Título:	Evitar monitorización propia.
Proviene de:	CAP-02, CAP-03, CAP-04
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema no debe interceptar las invocaciones generadas por su propia funcionalidad.

Tabla 30: RN-01

Identificador: RN-02	
Título:	Funcionamiento habitual no alterado.
Proviene de:	RES-09
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	La interceptación de los flujos de datos del proceso monitorizado no alterará la rutina de tratamiento habitual de los mismos.

Tabla 31: RN-02

3.4.3.3. Requisitos No Funcionales

Identificador: RNF-01	
Título:	Entorno de análisis y almacenamiento.
Proviene de:	RES-12
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El entorno de análisis y almacenamiento será un PC.

Tabla 32: RNF-01

Identificador: RNF-02	
Título:	Entorno de monitorización.
Proviene de:	RES-08
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El entorno de monitorización será un dispositivo real con sistema operativo Android con versión Android OS versión v4.4 KitKat o superior.

Tabla 33: RNF-02

Identificador: RNF-03	
Título:	Dispositivo con acceso a internet.
Proviene de:	RES-15
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El dispositivo debe contar con conexión a Internet.

Tabla 34: RNF-03

Identificador: RNF-04	
Título:	Visibilidad entre dispositivo y PC.
Proviene de:	CAP-06
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El PC y el dispositivo móvil podrán comunicarse a través de la red, ya sea pública o privada.

Tabla 35: RNF-04

Identificador: RNF-05	
Título:	Nivel de la interceptación.
Proviene de:	RES-14
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	La interceptación de llamadas al sistema se realizará a nivel de Android nativo.

Tabla 36: RNF-05

Identificador: RNF-06	
Título:	Proceso para envío de datos.
Proviene de:	RES-10, RES-11
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El envío de datos al servidor lo realizará un proceso ajeno a la monitorización.

Tabla 37: RNF-06

Identificador: RNF-07	
Título:	Permiso para registrar servicios.
Proviene de:	RES-14
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El dispositivo Android debe permitir registrar servicios.

Tabla 38: RNF-07

Identificador: RNF-08	
Título:	Escalada de privilegios.
Proviene de:	RES-14
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El dispositivo Android debe permitir escalar a privilegios de supersusario.

Tabla 39: RNF-8

Identificador: RNF-09	
Título:	Integridad de los paquetes extraídos.
Proviene de:	RES-13
Prioridad:	☒ Alta ☐ Media ☐ Baja
Necesidad:	☒ Esencial ☐ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	La información monitorizada en el dispositivo debe llegar al servidor de forma íntegra y ordenada.

Tabla 40: RNF-09

Identificador: RNF-10	
Título:	Estructura modular.
Proviene de:	RES-16
Prioridad:	☐ Alta ☒ Media ☐ Baja
Necesidad:	☐ Esencial ☒ Deseable ☐ Opcional
Verificabilidad:	☒ Si ☐ No
Estabilidad:	☒ Estable ☐ Inestable
Descripción:	El sistema tendrá una estructura modular que permita ampliar y mantener la funcionalidad en futuras versiones de Android.

Tabla 41: RNF-10

Capítulo 4: Diseño

En esta sección se define la arquitectura, los componentes, interfaces, y otras características del sistema. Además, se describen todas las decisiones de diseño tomadas con el fin de satisfacer los requisitos listados en la fase de análisis.

4.1. Visión General del Sistema

El diseño del sistema de monitorización de las llamadas al sistema se va a realizar para un dispositivo móvil sobre el que se ejecuta un sistema operativo Android, es decir, un entorno real, no simulado.

El sistema completo se compone de un conjunto de módulos operativos que van a actuar en cadena de forma que permitan:

- Asociarse en el proceso monitorizado.
- Interceptar las llamadas al sistema requeridas.
- Extraer los datos del proceso monitorizado hacia otro proceso corriendo en el dispositivo móvil.
- Enviar los datos a un servidor remoto y almacenarlos en un formato determinado para analizarlos.

Este diseño tendrá en cuenta en todo momento la portabilidad del sistema y su eficiencia en el uso de los recursos reducidos con los que cuenta un dispositivo móvil. Para ello, la monitorización se realizará en espacio de usuario, sin necesidad de modificar el *Kernel* del sistema operativo ni instalar aplicaciones nativas adicionales. Esto permite que, en caso de querer desplegarlo en distintos dispositivos, sean necesarios los mínimos cambios posibles. Los permisos necesarios sobre el sistema operativo van a ser la escalada de privilegios a superusuario y el máximo nivel de permisividad de la política de seguridad de SELinux.

La monitorización se va a hacer desde una biblioteca dinámica implementada a nivel de Android nativo y cargada en el proceso objetivo para que intercepte sus llamadas al sistema. Dado que el proceso está en ejecución, va a ser necesario forzar la carga de la biblioteca. La funcionalidad de la biblioteca no va a ocuparse de extraer los datos del dispositivo, por motivos técnicos de funcionalidad y rendimiento. Para ello, la extracción la va a realizar otro proceso que ejecuta una aplicación Android, parte del sistema, y que va a enviar los datos a través de la red a un servidor de análisis, donde se van a interpretar y almacenar. La comunicación entre estos dos procesos se va a hacer a través de memoria compartida para evitar utilizar el mismo sistema de comunicación entre procesos que el que se está monitorizando y beneficiarse del excelente

rendimiento de una comunicación en memoria. Una vez recibidos, el servidor se encargará del almacenamiento y análisis de la información *"exfiltrada"*.

4.2. Definición de la Arquitectura del Sistema

En esta actividad se define la arquitectura general del sistema de información, especificando las distintas particiones físicas del mismo, la descomposición lógica en subsistemas de diseño y la ubicación de cada subsistema en cada partición, así como la especificación detallada de la infraestructura tecnológica necesaria para dar soporte al sistema de información.

4.2.1. Especificación del Entorno Tecnológico

Se ha definido el entorno tecnológico como las arquitecturas de soporte, tanto hardware como software, que va a requerir para su funcionamiento.

El sistema, tal y como ha sido diseñado estará formado por dos plataformas informáticas.

Plataforma	Hardware Requerido	Software Requerido
Dispositivo móvil	Dispositivo móvil (teléfono o tableta) con arquitectura ARM Cortex A7 o compatible y conexión de red.	Android OS versión v4.4 KitKat o superior con permisos de superusuario. Paquete BusyBox. Acceso a internet.
Servidor remoto	Ordenador convencional con al menos 4GB de RAM y 50GB de Disco y conexión de red.	Herramienta Android Debug Bridge (ADB). Intérprete de código Python. Acceso a internet.

Tabla 42: Entrono Tecnológico

Las interfaces de red de ambas plataformas pueden estar conectadas tanto a una red pública o Internet, como a una red privada, pero debe ser posible la comunicación entre ambas sin restricciones, siendo su única limitación el estado de la propia red.

El envío de paquetes de datos provenientes de la información extraída del proceso monitorizado en el dispositivo móvil se realizará a través de una comunicación establecida en esta red.

Esta separación hardware entre la estructura sobre la que se hace la monitorización y la estructura donde se lleva a cabo el análisis de los datos permite reducir en gran medida la carga de trabajo encomendada al dispositivo móvil, circunstancia que garantiza el funcionamiento habitual del dispositivo sin ser apreciable el aumento del tiempo de respuesta por parte del usuario. Además se evita la ocupación de espacio de almacenamiento adicional, pues en estos dispositivos es más limitado.

4.2.2. Descomposición en Subsistemas de Diseño

Los subsistemas de diseño empaquetan módulos operativos. Cada uno de los subsistemas que componen el sistema diseñado trabaja a distinto nivel dentro de la arquitectura donde se despliega. En este apartado se identifica el nivel concreto donde actúa cada uno de los subsistemas. Las dos arquitecturas con las que se trabajan son Android y un servidor remoto.

4.2.2.1. Android

Sobre la arquitectura Android se van a desplegar el módulo inyector de biblioteca dinámica, la biblioteca dinámica de monitorización y la aplicación de extracción de datos.

Subsistema Injector Module	
Subsistema	Injector Module.
Descripción	Para poder realizar la inyección de biblioteca dinámica requiere de la realización de llamadas al sistema y trabajo sobre direcciones de memoria. Para ejecutar algunas de estas invocaciones se requieren privilegios de superusuario.
Nivel de la arquitectura	Binaries.
Nivel de privilegios / Permisos	Superusuario.
Implementación	*C++*.
Modo de distribución	Ejecutable nativo en Android.

Tabla 43: Subsistema Injector Module

Subsistema LibHook	
Subsistema	Libhook.
Descripción	Monitoriza datos a nivel de llamada al sistema en espacio de usuario. Es necesaria la invocación de llamadas al sistema y trabajo sobre direcciones de memoria. SELinux podría bloquear la llamada *dlopen* invocada desde un proceso sin privilegios de superusuario.
Nivel de la arquitectura	Native Libraries / Native Code.
Nivel de privilegios / Permisos	Nivel de sensibilidad de SELinux 0.
Implementación	*C++*.
Modo de distribución	Biblioteca dinámica (*.so*).

Tabla 44: Subsistema Libhook

Subsistema Extraction APK	
Subsistema	Extraction APK
Descripción	El envío de datos al exterior de la APK puede realizarse desde capas de la arquitectura superiores, con los mecanismos de abstracción que estas ofrecen. Para comunicarse con la biblioteca dinámica va a necesitar ejecutar código nativo a través de la JNI. SELinux podría bloquear la el registro de servicios en el sistema desde un proceso sin privilegios de superusuario.
Nivel de la arquitectura	Application Framework / Native Code (JNI). Acceso a Internet.
Nivel de privilegios / Permisos	Nivel de sensibilidad de SELinux 0.
Implementación	Java / C++.
Modo de distribución	Empaquetado en APK (*.apk*).

Tabla 45: Subsistema Extraction APK

4.2.2.2. Servidor remoto

En el servidor remoto reside el subsistema de análisis de datos.

Subsistema Data Analysis Module	
Subsistema	Data Analysis Module
Descripción	Es necesaria la apertura de un socket de red y el acceso al sistema de ficheros.
Nivel de la arquitectura	Espacio de usuario.
Nivel de privilegios / Permisos	Acceso a Internet.
Implementación	Python.
Modo de distribución	Python File (*.py*).

Tabla 46: Subsistema Data Analysis Module

4.2.3. Identificación y Descripción de los Módulos del Sistema.

Es posible agrupar conjuntos de funcionalidades y componentes en módulos independientes con una función concisa. A continuación se describen los servicios que ofrece cada módulo y la manera desarrollar los mismos.

4.2.3.1. *Injector Module*

Este módulo contiene la lógica requerida para realizar una inyección de biblioteca dinámica en un proceso objetivo. Es decir, este módulo fuerza al proceso objetivo a cargar una biblioteca dinámica arbitraria en su espacio de direcciones.

Para ello se va a asociar al proceso utilizando la llamada al sistema *ptrace* y va a modificar sus registros para forzar la carga.

4.2.3.2. *Monitor Module*

Este módulo está contenido en la biblioteca dinámica, *libhook.so*. En ésta se va a desarrollar la funcionalidad para interceptar las llamadas al sistema descritas en el análisis mediante un parcheo de la PLT. Mediante esta técnica se van a suplantar (*hook*) las direcciones que el proceso objetivo tiene mapeadas de las llamadas al sistema originales por las direcciones de unas rutinas *hook* (una por llamada al sistema redirigida) donde se van a extraer los argumentos de cada invocación.

Estos argumentos se van a empaquetar como una cadena con distinto formato dependiendo de cada llamada al sistema, y se van a enviar al proceso de extracción de los datos a través del mecanismo que habilita el *Communication Module*.

4.2.3.3. *Communication Module*

Este módulo se encarga de habilitar una comunicación entre el *Monitor Module* y el *Extracting Module*, de forma que los datos que se obtienen al monitorizar sean empaquetados y enviados a otro proceso que se encargue de exfiltrarlos del dispositivo.

El tipo de comunicación entre procesos a bajo nivel que habilita este módulo es la compartición de un fichero en memoria.

Está compuesto por una parte cliente, la cual reside junto al *Monitor Module* en la biblioteca dinámica inyectada en el proceso monitorizado, *libhook.so*. Y una parte servidor que reside junto al *Extracting Module* en la *Extraction APK*.

El cliente va a abrir un fichero en el componente Android que habilita la compartición de regiones de memoria, *ashmem*, y mapearlo en memoria. El cliente tiene que compartir el

descriptor de fichero mapeado para que el proceso que ejecuta la *Extraction APK*, pueda mapearlo también y notificar instantáneamente los cambios en memoria al *Extracting Module*.

La compartición del descriptor de fichero se va a realizar a través del Android Binder. La parte servidor va a registrar un servicio nativo en el sistema que ofrezca una función que mapee en memoria el descriptor de fichero que reciba. El cliente tendrá que consumir esa función remota para compartir el descriptor de fichero. Es importante recalcar que esta interacción mediante el Binder ocurre antes de que la monitorización del proceso comience, y por lo tanto no va a alterar los eventos de análisis normales ya que no va a quedar registrada en los datos extraídos al servidor.

La parte cliente está implementada en C/C++ como parte de la biblioteca inyectada. La parte servidor tiene componentes en Java y en código nativo (C/C++) que se comunican entre sí mediante JNI.

4.2.3.4. *Extracting Module*

Es el encargado de exfiltrar los datos que recibe del módulo de comunicación a través de la red con destino a un servidor conocido, donde se almacenarán y analizarán. Reside en la parte de alto nivel de la *Extraction APK* y está implementado enteramente en Java.

El mecanismo de comunicación que van a utilizar el *Communication Module* y el *Extracting Module* va a ser a través de una doble referencia a un objeto *Handler* instanciado en cada proceso. Este objeto está asociado al hilo y a su cola de mensajes. Permite al proceso enviar y procesar mensajes. Las peticiones tienen forma de *Message* y se resuelven a través de Android Binder.

Este módulo se ejecuta como la actividad principal de la aplicación, por lo que no tiene permitido ejecutar llamadas bloqueantes. La actividad de red la ejecutará mediante una tarea asíncrona.

4.2.3.5. *Data Analysis Module*

En este módulo se llevará a cabo el estudio del comportamiento del proceso objetivo con el fin de detectar patrones de comportamiento. Es el único que reside en el servidor remoto.

En su versión actual, únicamente obtiene estadísticas de las llamadas al sistema y de los eventos de IPC más utilizados por la aplicación monitorizada, tanto en términos absolutos como en la evolución de las llamadas a lo largo del tiempo.

De esta forma, permite conocer cuál es la dinámica usada por las aplicaciones y su interacción con el *Kernel* y con otros procesos. Precisamente el carácter modular del diseño permite que este módulo pueda ser modificado en un futuro para enriquecer el análisis de datos, por ejemplo haciendo uso de técnicas de minería de datos o aprendizaje automático que permitan reconocer patrones y ayude a la detección de aplicaciones maliciosas.

4.2.3.6. *Launcher Module*

Este módulo consiste en un *script* de activación del sistema. Se encarga de activar la Extraction APK y el Injector Module, además de enviar metadatos sobre el proceso objetivo al servidor remoto.

4.2.4. Diagrama de Despliegue o Distribución

Un diagrama de despliegue modela la distribución física de los objetos dentro de nodos. En él se representan los componentes hardware existentes (nodos), los componentes software que se ejecutan sobre cada nodo (objetos) y los tipos de conexión entre las diferentes piezas [19].

La Figura 12 muestra el diagrama de distribución del dispositivo móvil y el ordenador convencional como nodos hardware. El *Injector Module*, el subsistema *Libhook*, con sus módulos *Monitor Module* y *Communication Module*, y el subsistema *Extraction APK* con sus módulos *Extraction Module* y *Communication Module*, residen en el dispositivo móvil. El *Data Analysis Module* es el único componente software que reside en el ordenador convencional. La conexión entre ellos se hace con la combinación de protocolos UDP/IP.

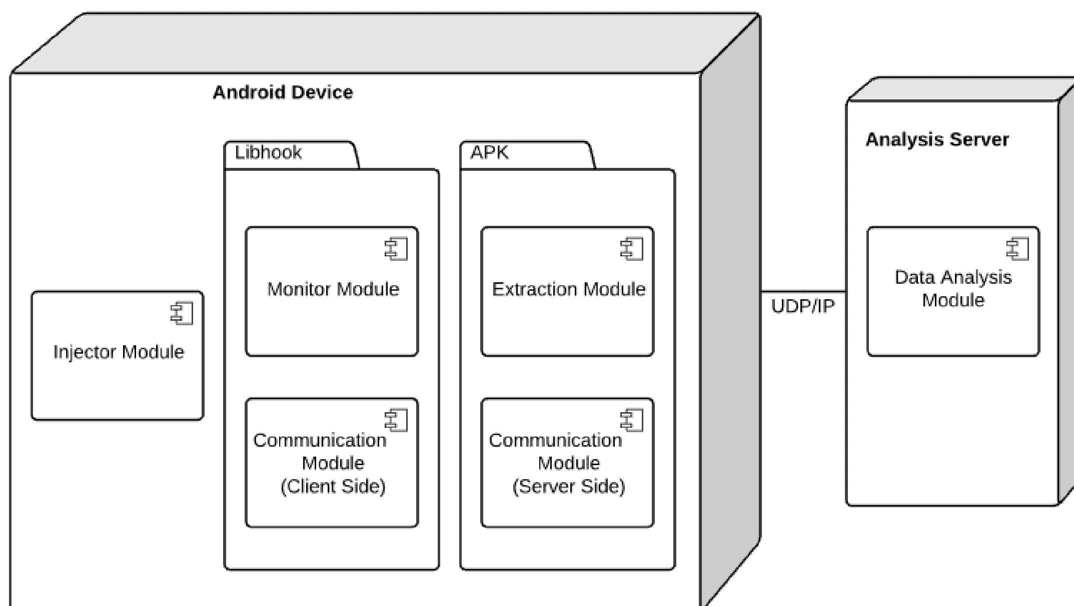


Figura 12: Diagrama de distribución

4.2.5. Diseño de la Arquitectura de Módulos del sistema

El objetivo de esta sección es definir los módulos del sistema de información, y la manera en que van a interactuar unos con otros, intentando que cada módulo trate total o parcialmente un proceso específico y tenga una interfaz sencilla.

4.2.5.1. *Diagrama de Componentes*

Un diagrama de componentes representa la composición del sistema a través de sus componentes y las conexiones entre ellos. Se utiliza para ilustrar la estructura de sistemas complejos [19]. La Figura 13 muestra el diagrama de componentes de la aplicación desarrollada.

Los paquetes de funcionalidad dentro de cada módulo son representados mediante componentes. El cableado muestra las dependencias directas de unos componentes sobre otros y el consumo de interfaces por parte de otros componentes.

Se ha logrado una descomposición modular eficaz pues las dependencias entre módulos delegan en la dependencia de únicamente un componente del módulo dependiente sobre un componente del módulo dependido.

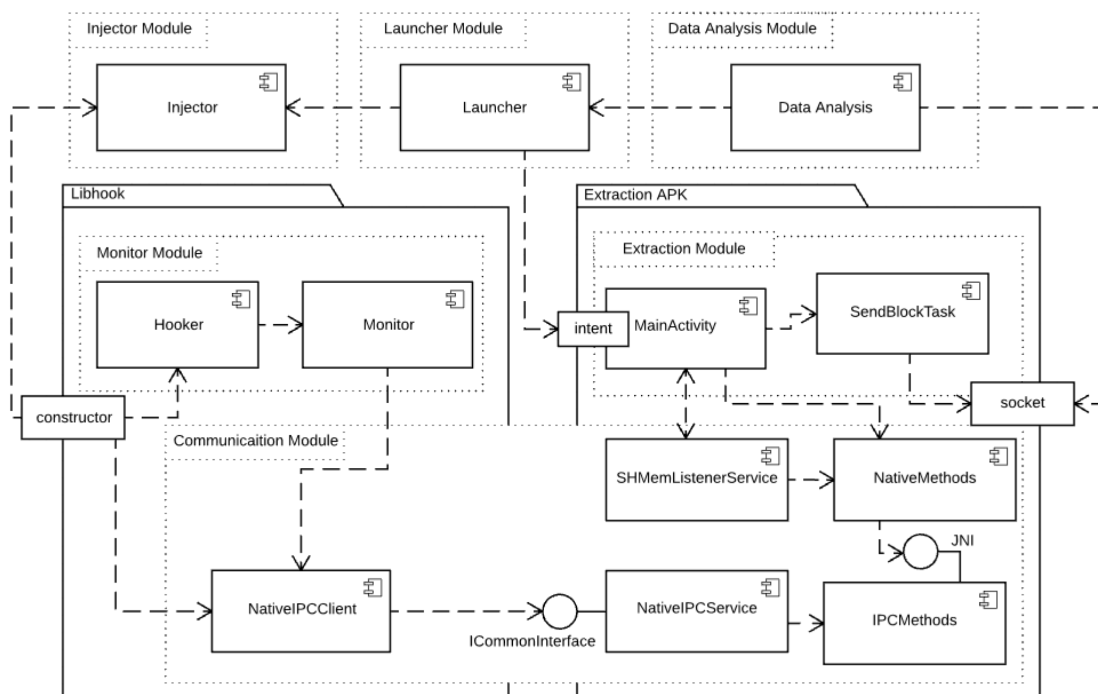


Figura 13: Diagrama de componentes

4.2.5.2. *Modelo de Dependencias Entre Componentes*

Identificador: DEP-01	
Componente origen	Hooker.
Componente objetivo	Monitor.
Técnica	Fichero de cabecera C/C++ (*.h*).
Descripción	Conjunto de rutinas *hook* listadas en el fichero de cabecera del *Monitor Component*.

Tabla 47: DEP-01

Identificador: DEP-02	
Componente origen	Monitor.
Componente objetivo	NativeIPCClient.
Técnica	Zona de memoria.
Descripción	Puntero a memoria compartida habilitado por NativeIPCClient al mapear un fichero en memoria compartida.

Tabla 48: DEP-02

Identificador: DEP-03	
Componente origen	NativeIPCClient.
Componente objetivo	NativeIPCService.
Técnica	Interfaz nativa de tipo *ICommonInterface* ofrecida por NativeIPCService compartida entre ambos componentes.
Descripción	La interfaz de tipo *ICommonInterface* contiene la declaración de las funcionalidades remotas que ofrece el servicio NativeIPCService a nivel nativo.

Tabla 49: DEP-03

Identificador: DEP-04	
Componente origen	NativeIPCService.
Componente objetivo	IPCMethods.
Técnica	Biblioteca nativa.
Descripción	La implementación de los métodos que ofrece NativeIPCService se encuentra en el código nativo contenido en IPCMethods.

Tabla 50: DEP-04

Identificador: DEP-05	
Componente origen	NativeMethods.
Componente objetivo	IPCMethods.
Técnica	JNI.
Descripción	NativeMethods contiene la declaración de los métodos Java que referencian a métodos nativos implementados en IPCMethods, a los que accede a través de la JNI.

Tabla 51: DEP-05

Identificador: DEP-06	
Componente origen	NativeMethods.
Componente objetivo	IPCMethods.
Técnica	JNI.
Descripción	NativeMethods contiene la declaración de los métodos Java que referencian a métodos nativos implementados en IPCMethods, a los que accede a través de la JNI.

Tabla 52: DEP-06

Identificador: DEP-06	
Componente origen	SHMemListenerService.
Componente objetivo	NativeMethods.
Técnica	Clase Java.
Descripción	SHMemListenerService accede a la funcionalidad nativa a través de los métodos declarados en NativeMethods.

Tabla 53: DEP-06

Identificador: DEP-07	
Componente origen	MainActivity.
Componente objetivo	NativeMethods.
Técnica	Clase Java.
Descripción	MainActivity accede a la funcionalidad nativa a través de los métodos declarados en NativeMethods.

Tabla 54: DEP-07

Identificador: DEP-08	
Componente origen	MainActivity.
Componente objetivo	SHMemListenerService.
Técnica	Objeto Handler de Java.
Descripción	MainActivity obtiene una referencia a SHMemListenerService utilizando un objeto Handler que está asociado al hilo y a la cola de mensajes de ese hilo.

Tabla 55: DEP-08

Identificador: DEP-09	
Componente origen	SHMemListenerService.
Componente objetivo	MainActivity.
Técnica	Objeto Handler de Java.
Descripción	SHMemListenerService obtiene una referencia a MainActivity utilizando un objeto Handler que está asociado al hilo y a la cola de mensajes de ese hilo.

Tabla 56: DEP-09

Identificador: DEP-10	
Componente origen	MainActivity.
Componente objetivo	SendBlockTask.
Técnica	Objeto AsyncTask de Java.
Descripción	MainActivity invoca una tarea asíncrona utilizando un objeto AsyncTask.

Tabla 57: DEP-10

4.2.5.3. Especificación de Componentes

En esta sección se va a desarrollar el diseño de los componentes software de una manera orientada a la construcción del sistema. Para ello se va a tomar como referencia la descripción de los módulos y los requisitos funcionales y no funcionales listados en el análisis del sistema.

Identificador: CO-01	
Componente	Injector.
Módulo	Injector Module.
Descripción	Ejecutable implementado en C++.
Definición	Contiene la lógica necesaria para inyectar una biblioteca dinámica con extensión *.so* en un proceso objetivo.
Ubicación	Acceso a Internet.
Dependencias	Ninguna.
Procedimiento	1. Asociarse al proceso objetivo con la llamada al sistema *ptrace*. 2. Resolver las direcciones de las funciones remotas que es necesario que el proceso objetivo invoque. (Calcular desplazamiento de ASLR) 3. Modificar los registros del proceso en ejecución para que ejecute las funciones remotas que permiten cargar la biblioteca dinámica.

Tabla 58: CO-01

Identificador: CO-02	
Componente	Hooker.
Módulo	Monitor Module.
Descripción	Componente de código C++ ejecutado por el constructor de la biblioteca dinámica.
Definición	Hace un *patching* de la PLT del proceso para redirigir las llamadas al sistema listadas en el fichero de cabecera del *Monitor Component*.
Ubicación	Funcionalidad dentro de *libhook.so*.
Dependencias	DEP-01.
Procedimiento	1. Obtener una lista de todos los módulos cargados por el proceso. 2. Obtener la dirección virtual base de cada módulo utilizando *dlopen* sobre ellos. 3. A partir del puntero, buscar el símbolo original de cada llamada al sistema dentro de su *sym table*. 4. Buscar el símbolo dentro de la PLT. 5. Guardar la dirección original de la llamada al sistema y sustituir la dirección por la de las rutinas *hook* del Monitor Module.

Tabla 59: CO-02

Identificador: CO-03	
Componente	Monitor.
Módulo	Monitor Module.
Descripción	Conjunto de rutinas *hook* en código C++ invocadas cuando el proceso objetivo invoca la llamada al sistema redirigida.
Definición	Cada rutina *hook* implementada está declarada de forma idéntica a su correspondiente función de llamada al sistema, pero tienen distinto comportamiento. Cuando una rutina *hook* es invocada, extrae los parámetros que recibe para escribirlos tal y como está indicado en el Modelo de Datos en la zona de memoria compartida habilitada por NativeIPCClient. Faltan funciones
Ubicación	Funcionalidad dentro de *libhook.so*.
Dependencias	DEP-02.
Procedimiento	Para cada rutina *hook* invocada: 1. Invoca a la llamada al sistema original pasándole los mismos parámetros que recibe. 2. Forma un mensaje con la estructura que indique el Modelo de datos, dependiendo de cada rutina. 3. Escribe el mensaje en memoria compartida siguiendo el protocolo de lectura y escritura. 4. Retorna el valor resultante de invocar la función original.

Tabla 60: CO-03

Identificador: CO-04	
Componente	NativeIPCClient.
Módulo	Communication Module.
Descripción	Conjunto de clases nativas que implementan un cliente nativo que consume un servicio Android. Implementado en C++.
Definición	Mapea un fichero en memoria compartida e invoca un servicio remoto al que envía el descriptor de fichero mapeado para habilitar la comunicación a través del fichero mapeado en memoria. Utiliza las clases que ofrece la biblioteca nativa de Android para establecer IPC a través del *Binder Kernel Driver*.
Ubicación	Funcionalidad dentro de *libhook.so*.
Dependencias	DEP-03.
Procedimiento	1. Obtiene una referencia al servicio remoto a través de su nombre y una interfaz común. 2. Abre un fichero en la zona de compartición de memoria de Android (*/dev/ashmem*). 3. Mapea el descriptor de fichero en memoria de forma que otros procesos puedan mapearlo y observar los cambios. 4. Comparte el descriptor de fichero invocando las funciones que ofrece la interfaz común a través de la referencia al servicio. 5. Almacena el puntero a memoria compartida.

Tabla 61: CO-04

Identificador: CO-05	
Componente	NativeIPCService.
Módulo	Communication Module.
Descripción	Servicio implementado en código nativo y empaquetado en la Extraction APK.
Definición	Registra un servicio en el sistema, el cual ofrece una función remota de mapeo de fichero en memoria compartida a través de una interfaz común que implementan cliente y servicio. Utiliza las clases que ofrece la biblioteca nativa de Android para establecer IPC a través del Binder Kernel Driver.
Ubicación	Biblioteca nativa empaquetado en Extraction APK.
Dependencias	DEP-04.
Procedimiento	1. Se registra el servicio en el sistema con un nombre identificativo. 2. Queda a la espera de ser invocado. 3. Cuando se activa por una petición, mapea el descriptor de fichero que recibe en el espacio de direcciones del proceso. 4. Almacena el puntero a memoria compartida.

Tabla 62: CO-05

Identificador: CO-06	
Componente	NativeMethods.
Módulo	Communication Module.
Descripción	Declaración Java de funcionalidad nativa. Clase Java.
Definición	Clase que contiene las referencias Java a los métodos nativos a los que se accede a través de la JNI. Clase Java.
Ubicación	Clase Java empaquetada en Extraction APK.
Dependencias	DEP-05.
Procedimiento	Carga y referencia a los métodos nativos que se invocan directamente desde Java.

Tabla 63: CO-06

Identificador: CO-07	
Componente	SHMemListenerService.
Módulo	Communication Module.
Descripción	Servicio Android. Clase Java de tipo IntentService.
Definición	Monitoriza la región de memoria compartida a través de un método nativo y envía los nuevos paquetes al Componente MainActivity a través de un mecanismo de IPC.
Ubicación	Empaquetado en Extraction APK.
Dependencias	DEP-06, DEP-09.
Procedimiento	1. Se registra el servicio en el sistema. 2. Obtiene la referencia a MainActivity a través de un Handler, para el envío de mensajes. 3. Monitoriza la zona de memoria compartida a través de métodos nativos a la espera de cambios. 4. Si notifica la llegada de un paquete se eleva de código nativo a Java y se envía a la actividad principal mediante el mecanismo de IPC.

Tabla 64: CO-07

Identificador: CO-08	
Componente	MainActivity.
Módulo	Extracting Module.
Descripción	Actividad principal de la APK. Clase Java de tipo Activity.
Definición	Inicializa todos los componentes de la APK y espera a la recepción de paquetes para enviarlos a la red.
Ubicación	Empaquetado en Extraction APK.
Dependencias	DEP-07, DEP-08, DEP-10.
Procedimiento	1. Registra el servicio SHMemListenerService en el sistema. 2. Habilita obtiene la referencia al servicio SHMemListenerService mediante un *Handler*. 3. Registra el servicio IPCNativeService en el sistema a través de un método nativo. 4. Si recibe un paquete de SHMemListenerService, invoca una tarea asíncrona SendBlockTask para que la envíe a la red.

Tabla 65: CO-08

Identificador: CO-09	
Componente	SendBlockTask.
Módulo	Extracting Module.
Descripción	Tarea de invocación asíncrona. Clase Java de tipo AsyncTask.
Definición	Recibe un bloque de datos que envía a un servidor destino.
Ubicación	Empaquetado en Extraction APK.
Dependencias	Ninguna.
Procedimiento	1. Es instanciada con una dirección IP y PORT destino. 2. Al invocarse recibe un bloque de datos. Abre un socket de red y se envía por protocolo UDP/IP a la dirección destino.

Tabla 66: CO-09

Identificador: CO-10	
Componente	Data Analysis.
Módulo	Data Analysis Module.
Descripción	*Script* escrito en *Python*.
Definición	Procesa el fichero de datos (*.csv*) almacenado en el servidor y realiza tareas de análisis.
Ubicación	Servidor remoto.
Dependencias	Ninguna.
Procedimiento	1. Interpreta y procesa la información almacenada. 2. Realiza un análisis de frecuencia de los eventos monitorizados. 3. Obtiene datos de los eventos más frecuentes y de la evolución de los eventos a lo largo del tiempo. 4. Proporciona gráficas para la visualización de resultados.

Tabla 67: CO-10

Identificador: CO-11	
Componente	Launcher
Módulo	Launcher Module
Descripción	*Script* escrito en *Python*.
Definición	Inicializa todos los subsistemas.
Ubicación	Ordenador convencional.
Dependencias	Ninguna.
Procedimiento	1. Activa la Extraction APK enviando un *Intent* de activación a MainActivity 2. Envía al servidor remoto el contenido de la tabla */proc/<PID>/fd/* del proceso objetivo. 1. Activa el *injector* y le pasa como argumentos el identificador del proceso objetivo (PID) y el nombre de enlazado de la biblioteca a inyectar.
Argumentos de entrada	1. Arg1: <…/injector> (*Pathname* del componente Injector). 2. Arg2: <PID> (Identificador del proceso objetivo.) 3. Arg3: <…/libhook.so> (Nombre de enlazado de la biblioteca a inyectar). 4. Arg4: <IP> (Dirección IP del servidor remoto). 5. Arg5: <PORT> (Puerto de escucha del servidor remoto). 6. Arg6: <ACTIVITY> (*Pathname* de la actividad principal de la Extraction APK)

Tabla 68: CO-11

4.3. Diseño Físico de Datos

En esta sección se describe el formato de los paquetes de datos generados durante la monitorización y enviados a través de la red hacia el servidor de análisis.

4.3.1. Estructura de Paquete

Cada paquete enviado tiene una estructura predefinida, donde los primeros 4 *bytes* son el identificador de paquete. A continuación de ese identificador, irán concatenados mensajes hasta completar los 2048 *bytes*. En caso de que los datos no permitan ajustarse a esta medida, se incorporará un relleno de caracteres nulos.

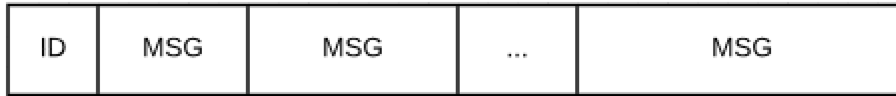


Figura 14: Estructura de un paquete

4.3.2. Estructura de Mensaje

Se genera un mensaje por cada llamada al sistema interceptada. El mensaje contiene una cabecera con metadatos que ocupa 14 *bytes* y un cuerpo de longitud variable dependiente de los argumentos que se recolecten de la invocación de la llamada al sistema en cuestión.

Figura 15: Estructura de mensaje

4.3.2.1. Estructura de Cabecera de Mensaje

La cabecera del mensaje tiene una estructura fija para todas las llamadas al sistema. Está compuesta por una etiqueta que indica el comienzo de un nuevo mensaje que ocupa 5 *bytes*, una marca temporal que ocupa 4 *bytes*, y el identificador del proceso que ha invocado la llamada, que ocupa 4 *bytes*. Cada campo está separado por un *byte* separador.

Figura 16: Estructura de cabezera de mensaje

4.3.2.2. Estructura del Cuerpo del Mensaje

El cuerpo de los mensajes tiene un tamaño variable que depende de los argumentos interceptados de cada llamada al sistema. La estructura del cuerpo de todas las llamadas al sistema contiene un prefijo que consiste en un código de 4 *bytes* que identifica la llamada al sistema.

Las llamadas regulares tendrán concatenados los argumentos con los que fueron invocadas. En caso de que el argumento sea un puntero o referencia a memoria, se copiará el contenido de esa zona de memoria referenciada.

Así, la estructura del cuerpo de las llamadas regulares queda como se muestra en la Figura 17. Cada campo está separado por un *byte* separador.

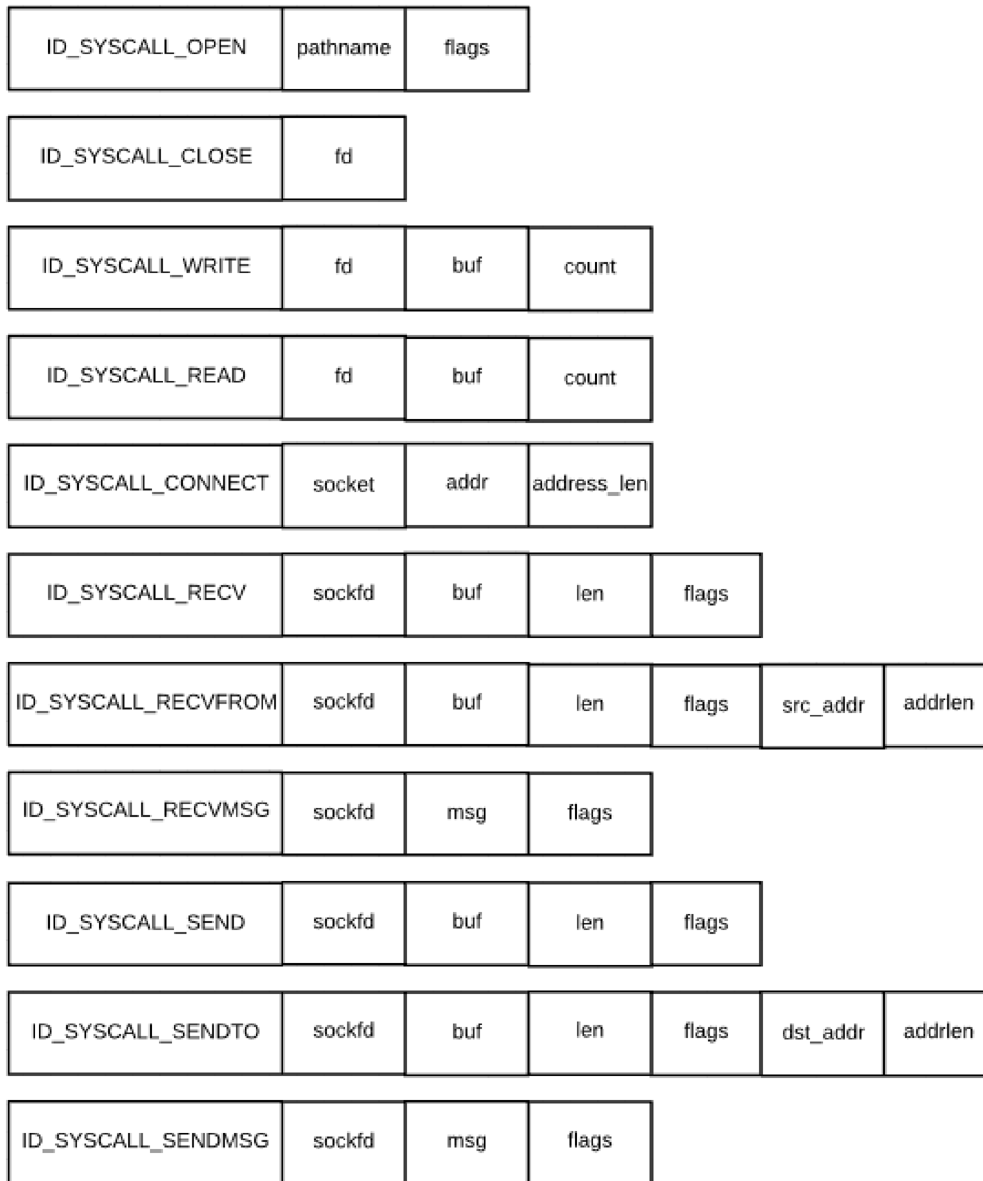


Figura 17: Estructura cuerpo llamadas regulares

La estructura del cuerpo generada por una llamada al Sistema *ioctl* va a contener los argumentos de la Binder Transaction interceptada. La estructura queda como se muestra en la figura Figura 18

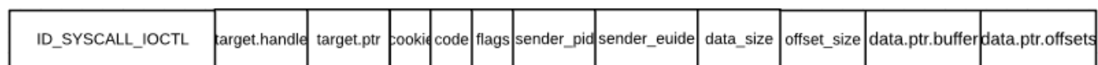


Figura 18: Estructura cuerpo llamada ioctl

4.3.3. Estructura de Paquetes de Mapeos de un Proceso

El mensaje enviado por el Launcher Module al Analysis Module que contiene los mapeos de los descriptores de fichero del proceso objetivo tiene un formato predefinido. Se incluye un prefijo con una etiqueta identificadora de este tipo de paquetes y a continuación se concatena las entradas de la lista de mapeos obtenida de */proc/<PID>/fd*.

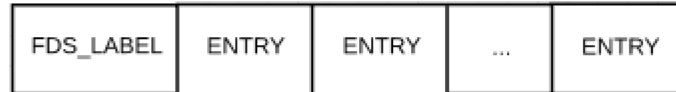


Figura 19: Estructura mensaje de mapeos

La estructura de cada entrada está formada por el descriptor de fichero y la ruta del mismo separados por un *byte* separador.

Figura 20: Estructura entrada

Capítulo 5: Implementación

5.1. Injector Module

5.1.1. Injector

Este módulo contiene la lógica requerida para inyectar una biblioteca dinámica en un proceso en ejecución. Consiste en un ejecutable implementado en C++.

Está basado en la implementación de Simone Margaritelli [9], y únicamente ha sido modificado para adaptarlo al presente Trabajo de Fin de Grado. Se puede obtener más información sobre detalles de la implementación en el Anexo 1 – Implementación de ARMINJECT.

5.2. Monitor Module

Habiendo analizado el funcionamiento del Android Binder, se conoce que la comunicación IPC en Android que se solicita al *Binder Kernel Driver* se lleva a cabo mediante la llamada al sistema *ioctl*. Además de esta, es necesario interceptar una serie de llamadas al sistema que pueden estar asociadas a IPC y enriquecen la recolección de datos.

5.2.1. Contructor de Libhook

El constructor de la biblioteca dinámica se ejecuta cuando esta es cargada por el proceso. En él se inicializan las estructuras donde se van a almacenar las direcciones originales de las rutinas de las llamadas al sistema, que apuntan a sus rutinas en *libc.so* (*bionic*), y las de las rutinas *hook* del Monitor a las que se les aplica la redirección de las llamadas originales.

En primer lugar se activa el cliente NativeIPCClient del Communication Module que devuelve la dirección de memoria virtual que referencia al fichero mapeado en memoria compartida. A continuación se invoca al Hooker para hacer el *hooking* de las llamadas al sistema de todos los módulos cargados por el proceso.

5.2.2. Hooker

Este componente reemplaza la dirección a la que apuntan las rutinas de las llamadas al sistema mapeadas originariamente en todos los módulos cargados por el proceso por direcciones de

funciones *hook* conocidas que comparten declaración. Se aplica para todas las rutinas de llamadas al sistema listadas en la estructura de *hooks* inicializada en el constructor.

Las direcciones de las rutinas *hook* las aporta el Monitor en su fichero de cabecera.

La implementación está basado en el código de Simone Margaritelli, que a su vez está basado en el código de Andrey Petrov. En este se realiza un patching de la PLT (Procedure Linkage Tables) de un proceso en ejecución. [9] [20]

5.2.3. Monitor

Consiste en un conjunto de rutinas de tratamiento de las *llamadas al sistema* redireccionadas por el Hooker, donde se invoca a la original y se recuperan los argumentos de la llamada antes de devolver retornar el valor al proceso que invoca. La Figura 21 muestra la relación de llamadas que se han interceptado, así como los argumentos de las mismas.

```
int hook_open(const char *pathname, int flags);
ssize_t hook_write(int fd, const void *buf, size_t len, int flags);
ssize_t hook_read(int fd, void *buf, size_t count);
int hook_close(int fd);
int hook_connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
ssize_t hook_send(int sockfd, const void *buf, size_t len, int flags);
ssize_t hook_sendto(int sockfd, const void *buf, size_t len, int flags,
                    const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t hook_sendmsg(int sockfd, const struct msghdr *msg, int flags);
ssize_t hook_recv(int sockfd, const void *buf, size_t len, int flags);
ssize_t hook_recvfrom(int sockfd, const void *buf, size_t len, int flags,
                    const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t hook_recvmsg(int sockfd, const struct msghdr *msg, int flags);
// NEW
int hook_ioctl(int fd, unsigned long request, char *argp);
```

Figura 21: Relación de llamadas interceptadas

Estas rutinas, tal y como estaban implementadas en el código original únicamente generaban un registro de *log* con los argumentos de invocación. En la implementación para este Trabajo se ha modificado la funcionalidad para extraer la información relevante de esos argumentos en mensajes empaquetados en bloques, que se escriben en el espacio de memoria compartida (*ashmem*) donde apunta el puntero que reporta NativeIPCClient siguiendo el Protocolo de Escritura en Memoria Compartida y el Diseño Físico de Datos de la fase de diseño.

Además, se ha añadido la rutina de la función *ioctl*, la cual no estaba incluida en el proyecto inicial de partida.

5.2.3.1. *hook_open*

Se recupera el *pathname* del fichero que se pretende abrir y el modo de apertura, los cuales se envían como una cadena, donde se incluye un identificador de llamada, a la función de empaquetado.

5.2.3.2. *hook_write*

Se recupera el nombre del fichero asociado al descriptor de fichero, la longitud del bloque a escribir y el modo de escritura, los cuales se envían como una cadena, donde se incluye un identificador de llamada, a la función de empaquetado.

5.2.3.3. *hook_read*

Se recupera el nombre del fichero asociado al descriptor de fichero, y la longitud del bloque de lectura, los cuales se envían como una cadena, donde se incluye un identificador de llamada, a la función de empaquetado.

5.2.3.4. *hook_close*

Se recupera el nombre del fichero asociado al descriptor de fichero que se quiere cerrar y se envía como una cadena, donde se incluye un identificador de llamada, a la función de empaquetado.

5.2.3.5. *hook_connect*

Se recupera el nombre del fichero asociado al descriptor de fichero del *socket*, la estructura que identifica al destino y la longitud de esta estructura y se envía como una cadena, donde se incluye un identificador de llamada, a la función de empaquetado.

5.2.3.6. *hook_send*

Se recupera el nombre del fichero asociado al descriptor de fichero del *socket*, el contenido del *buffer* de escritura, la longitud de este *buffer* y las *flags* y se envía como una cadena, donde se incluye un identificador de llamada, a la función de empaquetado.

5.2.3.7. *hook_sendto*

Se recupera el nombre del fichero asociado al descriptor de fichero del *socket*, el contenido del *buffer* de escritura, la longitud de este *buffer,* las *flags*, la estructura de dirección destino y su longitud y se envía como una cadena, donde se incluye un identificador de llamada, a la función de empaquetado.

5.2.3.8. *hook_sendmsg*

Se recupera el nombre del fichero asociado al descriptor de fichero del *socket*, el contenido del mensaje enviado y las *flags* y se envía como una cadena, donde se incluye un identificador de llamada, a la función de empaquetado.

5.2.3.9. *hook_recv*

Se recupera el nombre del fichero asociado al descriptor de fichero del *socket*, el contenido del *buffer* de lectura, la longitud de este *buffer* y las *flags* y se envía como una cadena, donde se incluye un identificador de llamada, a la función de empaquetado.

5.2.3.10. *hook_recvfrom*

Se recupera el nombre del fichero asociado al descriptor de fichero del *socket*, el contenido del *buffer* de lectura, la longitud de este *buffer,* las *flags*, la estructura de dirección origen y su longitud y se envía como una cadena, donde se incluye un identificador de llamada, a la función de empaquetado.

5.2.3.11. *hook_recvmsg*

Se recupera el nombre del fichero asociado al descriptor de fichero del *socket*, el contenido del mensaje recibido y las *flags* y se envía como una cadena, donde se incluye un identificador de llamada, a la función de empaquetado.

5.2.3.12. *hook_ioctl*

Este es el método más complejo, ya que no se limita a recuperar el valor de los argumentos, sino que se requiere recuperar datos apuntados por los punteros que contiene. Esto es porque la petición que una aplicación o proceso realiza al Android Binder para comunicarse con otro proceso se realiza a través de la llamada *ioctl* manejada por el *Binder Kernel Driver*. Este es el mecanismo en el que un proceso basa su IPC de forma directa.

La llamada *ioctl* se utiliza por convenio para realizar operaciones, comúnmente de I/O, que no se pueden expresar con las llamadas al sistema habituales o que no siguen un estándar, por lo que puede ser invocada con diversos fines que no sean IPC. Se conoce que las involucradas en IPC son las que cumplen una serie de requisitos, como se ha explicado en anteriormente.

```
#define BINDER_WRITE_READ          _IOWR('b', 1, struct binder_write_read)
#define BC_TRANSACTION             _IOW('c', 0, struct binder_transaction_data)
#define BC_REPLY                   _IOW('c', 1, struct binder_transaction_data)
```

Figura 22: Tipos de ioctl usados por el Binder

En su rutina *hook* se comprueba si el descriptor de fichero objetivo corresponde a un fichero que habilita la comunicación con el Binder Kernel Driver (*/dev/binder*), y en caso afirmativo hay que comparar el valor de la petición que se estaba realizando al Binder Kernel Driver. Si este valor coincide con el valor de *BINDER_WRITE_READ* (ver Figura 22), el cual es el principal comando de Android IPC frente a otros posibles, entonces se está iniciando una transacción con el Binder. El último argumento de la llamada se puede transformar en un puntero a una estructura de tipo *binder_write_read*, cuya declaración se encuentra en el fichero de cabecera de *binder.h* (ver Figura 3).

Esta estructura contiene principalmente dos componentes: un puntero a un buffer de escritura que almacenará la transacción con su código de operación y todos sus parámetros, la cual se envía al driver por el proceso, y además otro puntero a un buffer de lectura el cual contendrá la respuesta del driver tras ejecutarse la llamada al sistema *ioctl,* junto con el código que indica el tipo de respuesta.

Existen varios tipos de transacciones definidas en *enum binder_driver_command_protocol*, pero las que más información aportan con respecto a IPC son las transacciones que envían comandos, es especial la de tipo *BC_TRANSACTION*. Aunque también pueden estudiarse las BC_REPLY, este Trabajo se centra en las peticiones del proceso monitorizado, las cuales tienen el código mostrado en la Figura 23.

```
// BC_TRANSACTION code
const char bc_transaction_code[] = {0x00, 0x63, 0x28, 0x40};
```

Figura 23: Código de Binder Transaction

Cuando aparece el código de *BC_TRANSACTION* al comienzo del buffer de escritura de la estructura *binder_write_read*, se está lidiando con una petición de este tipo, por lo que la estructura que sucede al código de transacción es un *struct binder_transaction_data*, declarado en *binder.h*. Es la estructura clave en una Binder transaction, introducida en la sección 2.6.3 y mostrada en la Figura 4.

La estructura de la transacción contiene un *target*, el cual identifica el nodo Binder destino, realizando las distinciones entre si se trata de la invocación a un objeto en el espacio de memoria de un proceso remoto o en del proceso actual. El campo *cookie* está referido a información interna. El *sender ID* contiene información relevante sobre la seguridad de la comunicación. El campo *data* apunta a un buffer serializado en el que cada entrada se corresponde con un comando y los parámetros o argumentos que lo acompañan, los cuales son traducidos por la AIDL de los lados cliente y servidor. El campo *code* es el que indica el número de la función de la AIDL que se invoca, estando estas funciones numeradas de forma ordenada.

Todos estos datos que se recuperan de la estructura *binder_transaction_data,* se envían como una cadena separados por separadores, donde se incluye el identificador de la llamada *ioctl*, a la función de empaquetado.

5.3. Communication Module

Android habilita la compartición de memoria mediante su propio gestor de memoria, *ashmem*. El fichero creado en *ashmem* puede ser usado por diferentes procesos, pero el descriptor de fichero que se obtiene al abrirlo o el puntero resultante de mapearlo en memoria con *mmap* es dependiente del proceso.

La llamada al sistema *mmap* tiene que ser invocada sobre el mismo descriptor de fichero por todos los procesos que quieran compartir el fichero en memoria. En este caso, el proceso cliente debe crear y abrir el fichero en una zona del sistema de ficheros habilitada para la compartición de memoria como es */dev/ashmem*, mapear el fichero en memoria a través de su descriptor de fichero con *mmap* y compartir el descriptor de fichero con el proceso servidor para que este también pueda mapearlo. Para compartir este descriptor de fichero es necesario recurrir a otro mecanismo de IPC, el cual se va a realizar a través del Android Binder.

Para poder trabajar con el IPC que ofrece el Android Binder es necesario utilizar las clases que ofrecen las bibliotecas de Android. La clase *Binder*, que implementa la interfaz *IBinder* de Android consiste en una clase que representa a un objeto que se puede utilizar forma remota y transparente para el desarrollador. El servicio que registra el servidor va a ser tratado como un objeto remoto de tipo *Binder*.

La interfaz del cliente (*proxy*), de tipo *BpInterface*, y la del servidor (*stub*), de tipo *BnInterface,* implementan la interfaz común *ICommonInterface* donde se declaran las funciones que ofrece el servicio. El servidor, utilizando esa interfaz común como puerta de entrada, registra un servicio en el sistema que va a ser el que ejecute la lógica del mapeo en una de sus funciones. La interfaz del servidor va a actuar de la misma forma que una AIDL: el *stub* del lado del servidor se implementa con el método *onTransact*.

5.3.1. Componentes de la parte servidor – Extraction APK

La parte servidor reside en la APK y tiene dos funciones principales: habilitar la comunicación a través de memoria compartida con la parte cliente que reside en la biblioteca dinámica y controlar la llegada de paquetes desde el lado cliente, los cuales llegan en forma de cambios en el fichero mapeado en memoria. Estas dos tareas las asumen dos servicios respetivamente, los cuales está continuamente ejecutándose en segundo plano.

En la actividad principal de la APK se inicializan y se registran ambos servicios.

5.3.1.1. NativeMethods

Dentro de la APK se crea una clase para registrar los métodos que se van a ejecutar a través de JNI, la cual actúa como un punto de entrada a las funciones de dicha biblioteca. Esta clase se muestra parcialmente en la Figura 24. Además, esta clase carga el código nativo que se distribuye como una biblioteca dinámica.

```
public class NativeMethods {

    public static native boolean startService();
    public static native byte[] getBuffer();

    static {
        System.loadLibrary("ipc_methods");
    }
}
```

Figura 24: Clase de métodos nativos JNI

5.3.1.2. IPCMethods

Consiste en el código nativo empaquetado en la APK e implementado en C++. Se carga como una biblioteca.

En el código nativo, la función *JNI_OnLoad* se ejecuta en el momento en el que se carga la biblioteca nativa, por lo que es en esta función donde se van a enlazar los métodos listados en la parte java (NativeMethods), con las funciones implementadas en la parte nativa, tal y como muestra la Figura 25.

```
JNINativeMethod jniMethods[2] = {
        { "startService", "()Z", (void*)jni_startService},
        { "getBuffer", "()[B", (void*)jni_getBuffer},
};

jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv* env;

    if (vm->GetEnv(reinterpret_cast<void**>(&env), JNI_VERSION_1_6) != JNI_OK) {
        return -1;
    }

    jclass clz = env->FindClass("com/alex/nativeipcpoc/NativeMethods");
    env->RegisterNatives(clz, jniMethods, sizeof(jniMethods) / sizeof(JNINativeMethod));
    env->DeleteLocalRef(clz);

    return JNI_VERSION_1_6;
}
```

Figura 25: Implementación de los métodos nativos JNI

5.3.1.3. *NativeIPCService*

Este servicio es el que realiza la lógica de compartición de memoria. Su única tarea es ofrecer una función que se puede invocar de forma remota a través de la cual, recibiendo un descriptor de fichero de un fichero mapeado por el cliente en memoria compartida, realiza un mapeo sobre ese mismo descriptor de fichero teniendo acceso a esa zona de memoria física donde ambos procesos pueden hacer cambios visibles de forma instantánea.

Consiste en un servicio que se registra a nivel nativo en lenguaje C++ a través de la JNI, pues el tratamiento de la memoria es más sencillo trabajando en un lenguaje que tiene aspectos de bajo nivel como es C++.

Cuando se inicia la APK, desde la actividad principal se invoca al método de la JNI *startService*, que registra el servicio nativo en el sistema con un nombre identificativo. Para ello, desde el código nativo se instancia un objeto *Server* que implementa la interfaz *ServerInterface* la cual a su vez extiende a la interfaz *BpInterface* con referencia a *ICommonInterface*, ambas definidas en *binder/IInterface.h*. La traducción que tiene esta jerarquía de interfaces se explica a continuación:

- *ICommonInterface*: es la interfaz común que deben implementar cliente y servidor. En ella se declara el método remoto que va a ofrecer el servicio.
- *BnInterface*: es la interfaz del lado servidor. Se instancia como una referencia a *ICommonInterface* en el lado del servidor.
- *ServerInterface*: extiende de *BpInterface* e implementa el método *stub* del lado servidor, el cual es denominado *onTransact* en los entornos Android. Esta clase se muestra en la

Figura 26. Este es el método que se activa cuando se recibe una petición de un cliente y recibe un código de operación, que identifica la función a ejecutar por el servicio, un *Parcel* con los argumentos que pasa el cliente (unmarshalling), un *Parcel* para escribir la respuesta al cliente, y un campo *flags* que especifica el tipo de transacción del Binder Kernel Driver. Cuando resuelve el código, identifica la función a invocar y hace un unmarshalling de los datos que ha recibido para pasárselos como parámetros.

La única función que ofrece el servicio nativo que ha registrado el servidor es la de mapear un fichero en memoria compartida.

La función *shareFD* del objeto *Server* es la única función del servicio. Esta recibe como único argumento un descriptor de fichero, el que le pasa el método *onTransact* al hacer el *unmarshalling* del *Parcel* que envía el cliente. Si el descriptor de fichero es válido, es decir, si el cliente tiene abierto el fichero en su espacio de memoria, esta función ejecuta una llamada *mmap* sobre el mismo descriptor de fichero mapeando así el fichero en su espacio de memoria virtual. Si no ha habido ningún error, la dirección de memoria virtual que devuelve *mmap* apuntará a una zona de memoria física a la que el cliente está apuntando a su vez, pudiendo notificar cambios en memoria de forma instantánea. Esta dirección de memoria se almacena como una variable global para que se pueda acceder a ella en el ámbito de la biblioteca nativa.

```
// The server interface. Likewise the client interface, it also implements CommonInterface
// This class extends BpInterface (defined in <binder/IInterface.h>) with a reference to
// common interface
class ServerInterface : public BnInterface<ICommonInterface> {
    //this callback deals with the requests from the clients.
    // Everytime a transact() call is performed this method is executed in the server
    virtual status_t onTransact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0)
    {
        ALOGD("Server is dealing with a transact with code (%u)", code);

        //this checks if the interface exists and if the calle has propper permissions
        CHECK_INTERFACE(ICommonInterface, data, reply);
        //now we proceed depending on the requested method
        switch(code) {
            case SHARE_FD: {
                //in this case we send the result of dealing with the request
                reply->writeInt32(shareFD(data.readFileDescriptor()));
                return NO_ERROR;
            } break;

            default: {
                //code unknown, the server answers resending the transaction
                return BBinder::onTransact(code, data, reply, flags);
            } break;
        }
    }
};
```

Figura 26: Server Interface del módulo de comunicación

5.3.1.4. *SHMemListenerService*

Este servicio es el que notifica los cambios en la sección de memoria compartida. Consiste en un servicio que se registra en la capa Java de la APK, pero que ejecuta un método nativo en lenguaje C++ a través de la JNI, pues el tratamiento de la memoria es más sencillo trabajando en un lenguaje que tiene aspectos de bajo nivel como es C++.

El código nativo implemento en C++ pertenece a la misma biblioteca que el implementado para el servicio de compartición de fichero en memoria. Utiliza la misma clase para registrar su método *getBuffer*, el cual se ejecuta de forma nativa.

Cuando el servicio es iniciado por la actividad principal, se ejecuta su constructor, y a continuación se ejecuta su método *onHandleIntent* donde se implementa el código que el servicio va a ejecutar antes de finalizar. La función de este servicio es la de comprobar cambios en la zona de memoria compartida de forma ininterrumpida, por lo que su ejecución va a terminar cuando la aplicación se detenga.

El servicio invoca en este punto al método *chechNewMsg* donde entra en una espera activa en la que llama al método *getBuffer* de la JNI, el cual accede a su respectivo en la biblioteca nativa para que compruebe si hay cambios en memoria. El método *jni_getBuffer*, al residir en la biblioteca dinámica, tiene acceso a la dirección de memoria a la zona compartida. Este toma los cuatro primeros *bytes*, los cuales se corresponden con el identificador de paquete. Si el identificador cambia con respecto al de la última comprobación, reserva un espacio en memoria tan grande como indique la constante *CHUNK*, la cual debe coincidir en la parte cliente y servidor, y devuelve a Java el paquete que llega en forma de *byte array*. En este punto, el paquete se envía al hilo principal de la APK, donde reside el módulo de extracción. La forma de envío se va a explicar en el apartado del Extraction Module.

5.3.2. Componentes de la parte cliente - Libhook

5.3.2.1. NativeIPCClient

La parte cliente de este módulo crea y abre un fichero en una zona del sistema de ficheros visible para todos los procesos, mapea el fichero en memoria compartida a través de su descriptor de fichero y envía el descriptor de fichero al servicio levantado por el servidor para que este también pueda mapearlo.

Esta tarea se debe hacer en el momento en el que se carga la biblioteca dinámica inyectada, pues los datos interceptados por el Monitor Module van a ser empaquetados y compartidos en memoria durante la monitorización.

Como se ha comentado en la descripción del módulo anterior, cuando se carga la biblioteca dinámica se ejecuta su constructor, por lo que es en este donde se debe establecer la comunicación con el servicio. Así, lo primero que se hace dentro del constructor, es ejecutar la función *créate_shared_file()*. Los pasos que realiza el cliente son los siguientes:

- Se invoca la función auxiliar *getDemoServ()* para obtener una referencia al servicio (ver Figura 27). Para ello a través de un *ServiceManager* se obtiene una referencia al servicio remoto identificado con el nombre con el que ha sido registrado. El servicio se trata como un objeto remoto de tipo *Binder*. A continuación se instancia el objeto cliente,

BpCommonInterface que implementa la interfaz común *ICommonInterface*, pasándole como argumento el objeto remoto *Binder* que hace referencia al servicio. De esta forma se consigue un *proxy* que actúa como *stub* del cliente y se encarga de hacer el *marshalling* de los datos teniendo mapeadas las llamadas a los métodos remotos que ofrece el objeto *Binder*.

```
// Helper function to get a hold of the service.
sp<ICommonInterface> getDemoServ() {
    sp<IServiceManager> sm = defaultServiceManager();

    // service as a remote object
    sp<IBinder> binder = sm->getService(String16("NativeIPCService"));

    // instance of the client with reference to the service
    sp<ICommonInterface> demo = interface_cast<ICommonInterface>(binder);

    return demo;
}
```

Figura 27: Implementación del método getDemoServ

- Abre un fichero para compartir en */dev/ashmem*, y le asigna un tamaño fijo CHUNK (en experimentación 2048 *bytes*), y un nombre, de forma que pasa a llamarse */dev/ashmem/<NAME>*.
- Mapea el fichero entero en memoria compartida con *mmap* a partir de su descriptor de fichero. Habilita lecturas y escrituras de las páginas de memoria (*PROT_READ | PROT_WRITE*).
- Utilizando el objeto cliente, ejecuta la función *shareFD* de su *stub* (ver Figura 28), la cual recibe el descriptor de fichero, crea un *parcel* de envío y otro de respuesta, hace el *marshall* del descriptor de fichero en el *parcel* de envío e inicia una transacción con el servicio remoto enviando ambos *parcel* y además un código de operación que identifica a su función de mapeo de memoria compartida.

```
// ICommonInterface::shareFD implementation in BpCommonInterface (client-side)
status_t BpCommonInterface::shareFD(uint32_t fd){
    ALOGD("IPC Client Sharing File Descriptor from Client (%u)", fd);

    Parcel data, reply;
    //set the interface token i.e the name of the service that should deal with the request
    data.writeInterfaceToken(ICommonInterface::getInterfaceDescriptor());
    //write the file descriptor (marshall)
    data.writeFileDescriptor(fd);

    // start transaction
    remote()->transact(SHARE_FD, data, &reply);

    //the server puts the answer in reply, finally we return the result
    return reply.readInt32();
}
```

Figura 28: Implementación del método shareFD en el cliente

5.3.2.2. *Protocolos en Memoria Compartida*

<u>Mapeo de Fichero en Memoria Compartida.</u>

1. El cliente abre fichero en */dev/ashmem* con nombre *ashmem* y tamaño fijo *CHUNK* (en experimentación 2048 *bytes*).
2. El cliente ejecuta *mmap* sobre el descriptor de fichero para mapearlo entero en memoria habilitando lectura y escritura (*PROT_READ | PROT_WRITE*) y en modo *MAP_SHARED* para que todos vean los cambios.
3. El cliente instancia un *Parcel* donde incluye *marshalling* del descriptor de fichero e invoca al servicio remoto del lado servidor enviando el código de operación *SHARE_FD* para compartir el descriptor de fichero.
4. Se activa el método *onTransact* del *stub* del servicio al recibir la petición. Resuelve el código de operación e invoca a su función de mapeo en memoria compartida pasándole como argumento el descriptor de fichero, fruto del *unmarshalling* del *parcel*.
5. La función de mapeo del servicio comprueba si el descriptor de fichero es válido, y en caso afirmativo realiza sobre él un mapeo en memoria compartida *mmap* con las mismas características con las que lo ha realizado el cliente.
6. El *stub* del servicio da respuesta al cliente desbloqueando la espera de este.
7. El cliente y el servicio cuentan con una dirección virtual cada uno que se corresponde con la misma dirección física.

<u>Protocolo de Escritura en Memoria Compartida.</u>

Este protocolo se esquematiza en la Figura 29. Durante la inicialización desde el constructor de la biblioteca, el cliente reserva un espacio de memoria local con el mismo tamaño que el fichero compartido y escribe en los primeros 4 *bytes* el primer identificador de paquete.

- Se invoca una rutina *hook*.
  - La rutina genera el cuerpo del mensaje dependiente del Diseño Físico de Datos.
  - Se invoca la función de escritura en memoria pasándole como argumento el cuerpo del mensaje
    - Le incorpora la cabecera con el separador de mensaje, una marca de tiempo y el PID.
    - Si el mensaje cabe en el espacio de memoria auxiliar restante, se vuelca a continuación de los anteriores mensajes.
    - Si no cabe, se vuelca la zona de memoria auxiliar entera sobre la zona de memoria compartida, se incrementa el identificador de la zona de memoria auxiliar y se escribe el mensaje sobre la zona de memoria auxiliar vacía.

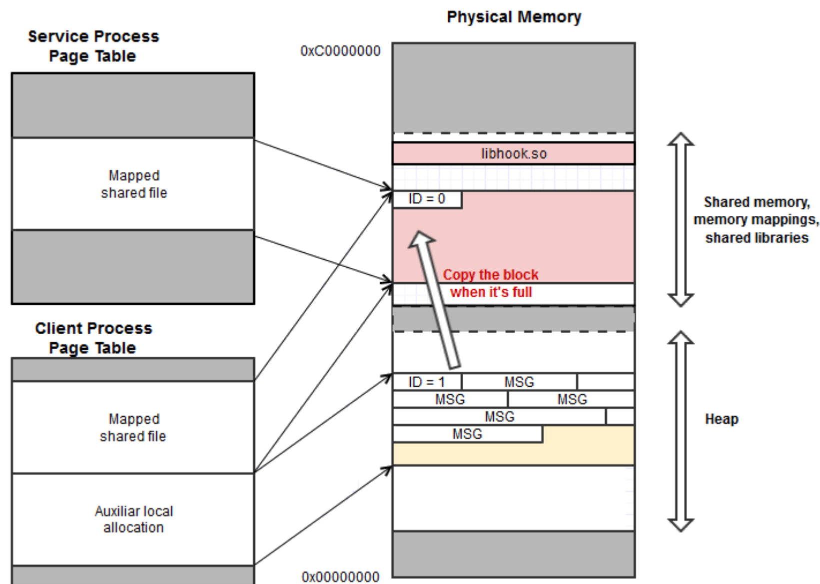


Figura 29: Esquema del protocolo de memoria compartida del módulo de comunicación

5.4. Extraction Module

La misión de este módulo es enviar los paquetes que recibe del módulo de comunicación a través de la red con destino a un servidor conocido, donde se almacenarán y analizarán los datos. De esta tarea se encarga el hilo principal de la aplicación.

La forma a través de la que se comunican el servicio SHMemListenerService, del Communication Module y el hilo principal de la aplicación, MainActivity, es a través de un Android *Messenger*.

5.4.1. Messenger IPC

Este objeto representa la referencia a un *Handler*, el cual puede ser enviado a un proceso remoto utilizando un *Intent*. El desarrollo sigue un modelo cliente servidor por suscripción.

Handler es un objeto asociado al hilo y a su cola de mensajes. Permite al proceso enviar y procesar mensajes. Las peticiones tienen el formato de *Message*, el cuál es similar a un *Intent,* y llevan asociado un código de operación (*Message.what*) y una carga útil (*Message.getData()*). El método *handleMessage()* se activa a la llegada de un mensaje, y hace la función de *stub*, resolviendo el código de operación y haciendo *unmarshalling* de los datos. El mecanismo de IPC que se utiliza por debajo es el Android *Binder*.

El servicio implementa un *Handler* para habilitar suscripciones. El hilo principal también implementa un *Handler* para resolver los *call-backs* del servicio con paquetes de datos.

Cuando la actividad principal inicia el servicio, guarda la referencia al mismo en un *Messenger* y se suscribe. La petición de suscripción es resuelta por el método *handleMessage()* del servicio, el cual guarda la referencia a la actividad principal en otro *Messenger*. Cada vez que el Communication Module obtenga un nuevo mensaje, *SHMemListenerService* va a lanzar un *call-back* a la actividad principal con el paquete de datos que haya recibido.

5.4.2. Envío de paquetes a la red

En Android el hilo principal de la APK no tiene permitido abrir conexiones de red. Esto está establecido así para evitar bloqueos en las operaciones de E/S durante la ejecución de la aplicación.

Para poder realizar estas operaciones, la actividad principal confía en *AsyncTask*. Esta clase permite realizar operaciones en un hilo en segundo plano y publicar los resultados en el hilo principal sin necesidad de manipular otros hilos o *Handlers*.

Cada vez que el servicio *SHMemListenerService* del módulo de comunicación lance un *call-back* con un paquete de datos, se va a instanciar un objeto de tipo *AsyncTask*, en este caso se ha definido en la clase *SendBlockTask*, asignando puerto y dirección IP destino. El método *AsyncTask.execute()* que se invoca desde la instancia activa el método *doInBackground* (Figura 30) de la clase, donde reside la lógica del envío de datos a la red. El envío se realiza por el protocolo UDP de la capa de transporte.

```
@Override
protected Boolean doInBackground(byte[]... params) {

    try {
        //instance udp socket
        DatagramSocket socket = new DatagramSocket();
        //set dst host address
        InetAddress address = InetAddress.getByName(hostName);
        // generate udp packet [buffer, length, host, port]
        DatagramPacket packet = new DatagramPacket(params[0], params[0].length, address, portNumber);
        //send
        socket.send(packet);
    } catch (Exception e){
        Log.e ("IPC_SendBlockTask", "Packet not sent");
    }
    return null;
}
```

Figura 30: Envío de paquetes por la red

5.5. Data Analysis Module

En este módulo se llevará a cabo el estudio del comportamiento del proceso objetivo con el fin de detectar actividades maliciosas. Es el único módulo que reside fuera del dispositivo móvil. Está implementado en Python. Consiste en un servidor que admite datagramas UDP que mantienen la estructura que establece el Diseño Físico de Datos.

Uno de los mecanismos de detección típicamente utilizado es el basado en anomalías. En él, primero se modela y analiza el comportamiento normal de aplicaciones que son “benignas”, es decir, que no contienen actividad maliciosa. Luego, en tiempo de detección, se contrasta la monitorización de eventos con ese modelo de normalidad para detectar desviaciones considerables y hacer saltar las alarmas. Es por ello que el módulo de análisis, en su versión actual, provee de datos estadísticos acerca de los eventos monitorizados. Concretamente, para cada instante *t*, calcula el número de eventos de cada tipo (*read, write, ioctl, etc.*) que se llevan ejecutados, con el fin de visualizar el comportamiento de la aplicación a lo largo del tiempo. Como se ha comentado anteriormente, este comportamiento podría luego compararse con otros para detectar desviaciones significativas que sean sospechosas.

La implementación está realizada en un script de Python, la cual hace uso de las librerías *csv* y *matplotlib* para la lectura de los ficheros en formato CSV y de la visualización de gráficos respectivamente. Por cada aplicación analizada, se muestra un histograma con las llamadas al sistema más usadas así como un diagrama de la evolución temporal de las mismas.

5.6. Launcher Module

Este módulo consiste en un *script* de activación del sistema.

Se invoca con los argumentos:

- Arg1: <…/injector> (*Pathname* del componente Injector).
- Arg2: <PID> (Identificador del proceso objetivo.)
- Arg3: <…/libhook.so> (Nombre de enlazado de la biblioteca a inyectar).
- Arg4: <IP> (Dirección IP del servidor remoto).
- Arg5: <PORT> (Puerto de escucha del servidor remoto).
- Arg6: <ACTIVITY> (*Pathname* de la actividad principal de la Extraction APK)

En primer lugar, a partir del identificador del proceso objetivo obtiene su tabla de descriptores de fichero */proc/<PID>/fd* para enviarle las entradas al servidor tal y como indica el Diseño Físico de Datos.

En segundo lugar activa la Extraction APK con el comando enviándole in *Intent* de activación a su actividad principal por línea de comandos.
Por último invoca al Injector Module para que inyecte la biblioteca dinámica en el proceso objetivo y comience la monitorización.

Capítulo 6: Verificación, Validación y Experimentación

6.1. Verificación del Sistema

En esta sección se describen las pruebas de verificación de requisitos realizadas.

El sistema generado solo cuenta con un caso de uso, por lo que mediante su mera activación se puede verificar si se han cumplido los requisitos casi en su totalidad.

Identificador: PV-01	
Nombre	Interceptación de llamadas al sistema.
Requisitos verificados	RF-01, RF-02, RF-03, RF-04, RF-05, RF-06, RF-07, RF-08, RF-09, RF-11, RNF-01, RNF-02, RNF-03, RNF-04, RNF-05, RNF-06, RNF-07, RNF-08, RNF-10.
Requisitos descartados	Ninguno.
Precondiciones	Ninguna.
Procedimiento	1. Ejecutar Launcher Module sobre el proceso de la aplicación *Google Chrome*. 2. Manipular la aplicación Google Chrome.
Resultados esperados	Recepción de paquetes por parte del servidor externo e impresión de trazas por pantalla.
Evaluación	Positiva.

Tabla 69: PV-01

Identificador: PV-02	
Nombre	Inspección estructura de paquete.
Requisitos verificados	RF-10, RF-12
Requisitos descartados	Ninguno.
Precondiciones	Ninguna.
Procedimiento	1. Ejecutar Launcher Module sobre el proceso de la aplicación *Google Chrome*. 2. Manipular la aplicación *Google Chrome.* 3. Observar estructura de paquetes volcados a fichero en el servidor de análisis.
Resultados esperados	Los mensajes tienen la estructura que especifica el Diseño Físico de Datos.
Evaluación	Positiva.

Tabla 70: PV-02

Identificador: PV-03	
Nombre	Pérdida de paquetes en red saturada.
Requisitos verificados	Ninguno.
Requisitos descartados	RNF-09.
Precondiciones	La red a través de la cual se comunican el dispositivo móvil y el servidor remoto es una red pública (*Eduroam*).
Procedimiento	1. Ejecutar Launcher Module sobre el proceso de la aplicación *Google Chrome*. 2. Manipular la aplicación *Google Chrome* cada 30 minutos durante 3 horas. 3. Observar el recuento de paquetes en el servidor de análisis en busca de la ausencia de algún identificador de paquete intermedio.
Resultados esperados	No falta ningún identificador de paquete.
Evaluación	Negativa. Ha habido pérdidas.

Tabla 71: PV-03

6.2. Extracción de Comportamientos

Para mostrar la utilidad de la herramienta desarrollada, se han realizado dos experimentos a modo de prueba de concepto. En estos experimentos, primero se ha analizado la actividad de la aplicación *Google Chrome* y de la Calculadora integrada de Android.

La prueba sobre la aplicación *Google Chrome* (PID:18489) se realiza una conexión con la página www.seg.inf.uc3m.es/publications.html y posteriormente se realiza la descarga de un fichero PDF, el cual se guarda en el dispositivo.

En la Figura 31 se puede observar cómo el mayor número de invocaciones (eventos) corresponden con la recepción (*recvfrom*) y la escritura en fichero (*write*) de los datos descargados, lo que es un comportamiento que cabe esperar.

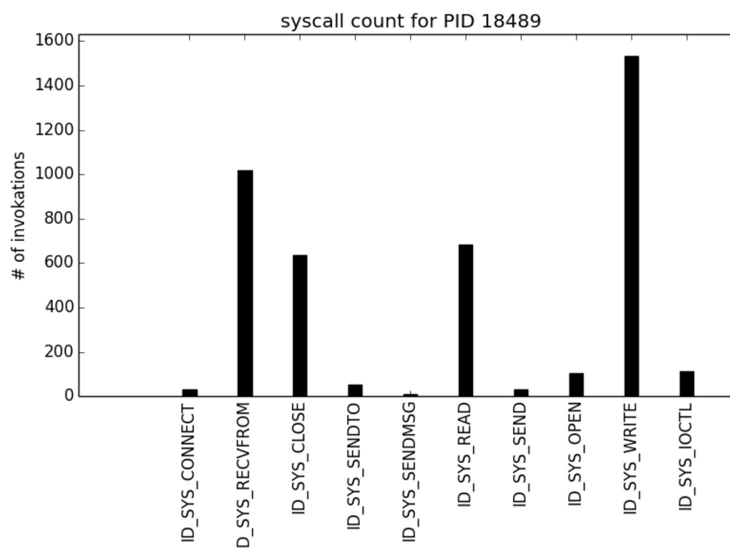


Figura 31: Eventos monitorizados para la app Google Chrome

En la Figura 32 se muestra la evolución de los eventos a lo largo del tiempo. Como se puede ver, se produce un incremento apreciable entre los segundos 12 y 15 de eventos *recvfrom*. Este periodo se corresponde con el momento en el que se procedido a la descarga del fichero PDF.

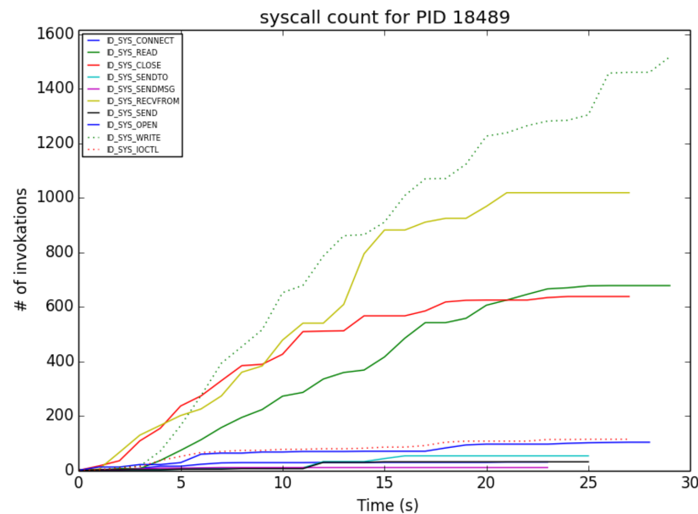


Figura 32: Evolución de eventos monitorizados para la app Google Chrome

La prueba sobre la aplicación de la calculadora (PID:9330) es muy básica. En ella se realiza una simple operación de suma de dos números y se cierra la aplicación. El objetivo de tan corto experimento es contar el número de eventos que una aplicación sencilla a priori pueda lanzar en un corto periodo de tiempo.

La Figura 33 muestra el agregado de eventos en los 4 segundos de duración del experimento, y la Figura 34 su evolución en el tiempo. Como se ve, hay más de 1200 eventos en total, los cuales corresponden a distintas llamadas. Lo que es más remarcable, es que se producen numerosos eventos de *recvfrom*, lo que podría hacer sospechar que la aplicación está recibiendo datos del exterior. Haciendo un análisis de estos eventos (es decir, de los argumentos de *recvfrom*), se puede deducir que la aplicación está utilizando *sockets* del sistema para comunicarse a nivel local, probablemente con otros componentes o hardware del dispositivo.

Esto da una idea de la dificultad de hacer un análisis dinámico de las aplicaciones, ya que estas tienen comportamientos que a priori pueden parecer sospechosos pero que realmente no lo son. Es por ello que disponer de herramientas que faciliten el análisis, como la desarrollada en este proyecto, es importante en la lucha contra el malware.

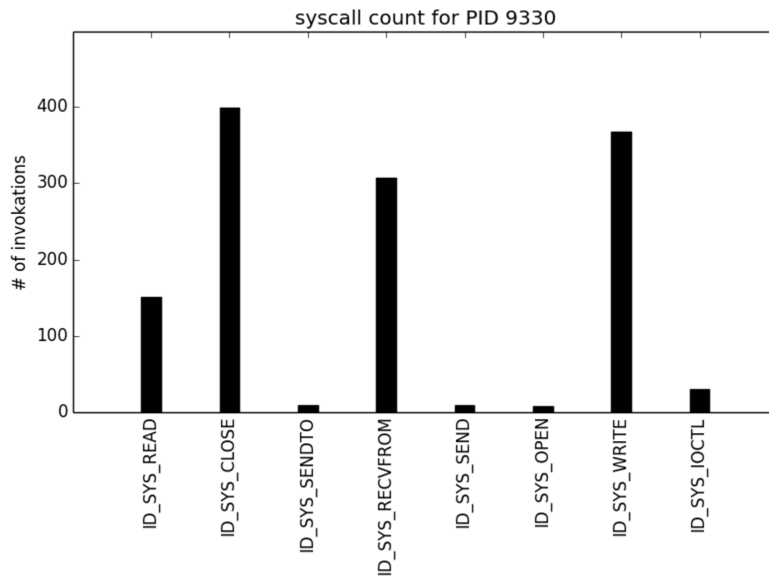


Figura 33: Eventos monitorizados para la app de la calculadora

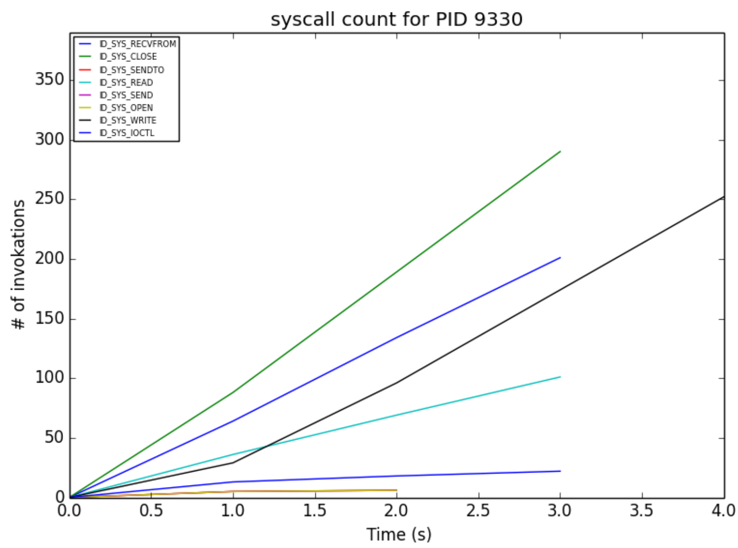


Figura 34: Evolución de los eventos monitorizados para la app de la calculadora

Capítulo 7: Planificación

Una buena planificación permite que los errores se detecten lo antes posible y por lo tanto, permite a los desarrolladores concentrarse en la calidad del software, en los plazos de implementación y en los costes asociados.

7.1. Ciclo de Vida del Proyecto

Los requisitos de usuario tienen un perfil muy técnico. En ellos se especifica con precisión todo lo que el sistema va a hacer pues existe documentación al respecto. Lo que diferencia al sistema es la innovación en el modo de hacerlo.

Esta circunstancia confiere al proyecto un carácter experimental, donde la eficacia depende en gran medida de la permisividad del sistema operativo. El rendimiento del sistema a desarrollar depende a su vez de las prestaciones que ofrezca la plataforma hardware sobre la que se trabaja. Por estas razones, existe la posibilidad de que los requisitos se tengan que redefinir en etapas avanzadas por cuestiones técnicas.

Teniendo en cuenta estas consideraciones se va a seguir un modelo de desarrollo incremental puro, tal como muestra la Figura 35, ya que proporciona flexibilidad en la toma de decisiones en base a los problemas encontrados durante el desarrollo.

Es un modelo basado en hitos que se centra en la entrega de un módulo operativo con cada incremento. Estos módulos pasan por todas las fases del desarrollo software. Implementan funcionalidades precisas que se irán agregando para formar el producto final.

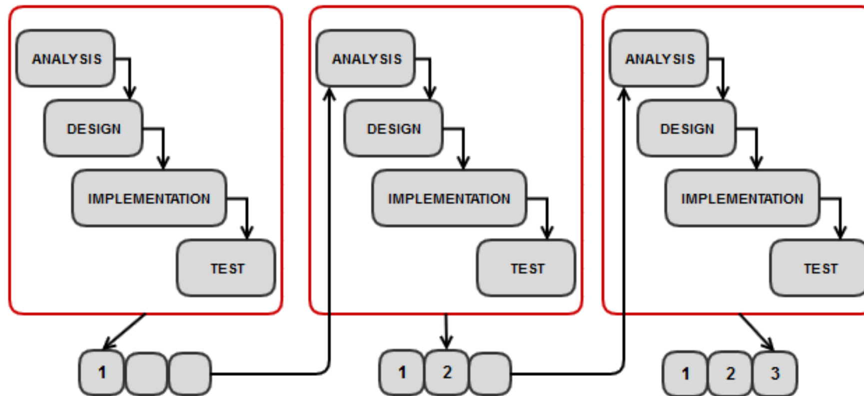


Figura 35: Ciclo de vida del proyecto

Este modelo es flexible. Al contar con el desarrollo de funcionalidades anteriores es más sencillo determinar si los requisitos para los subsiguientes niveles son correctos. Las decisiones de diseño con respecto a un módulo se pueden basar en el módulo anterior. De esta forma se reduce la incertidumbre de los procesos de análisis y diseño, por lo que se reduce el riesgo.

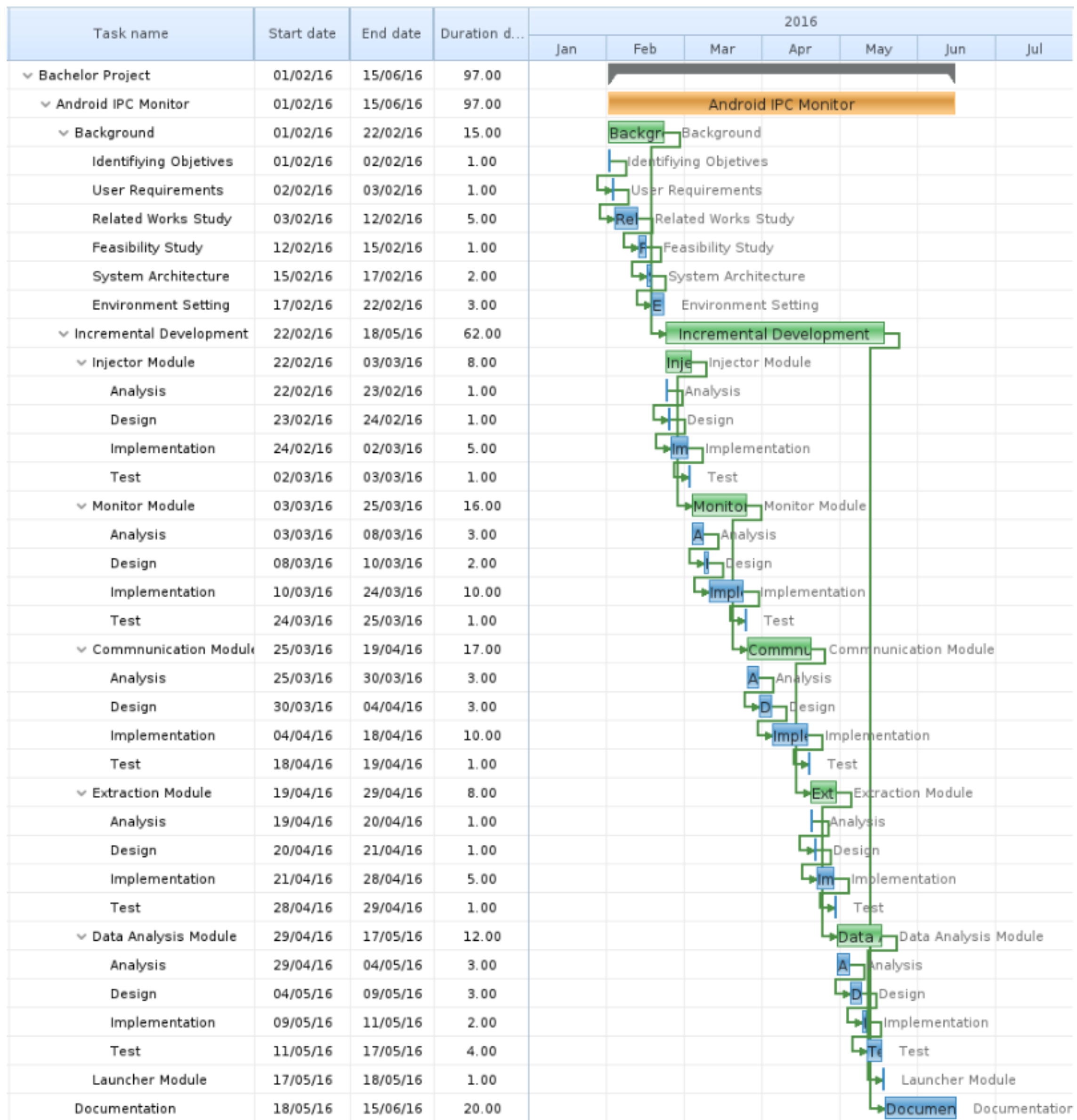
Mediante este modelo se genera software operativo de forma rápida y en etapas tempranas del desarrollo.

7.2. Diagrama de Gantt

Se recurre a un diagrama de Gantt para mostrar el tiempo de dedicación previsto para las diferentes tareas y actividades a lo largo del tiempo de desarrollo del proyecto.

La fecha de comienzo del proyecto es el 1 de febrero de 2016 y la fecha de finalización prevista es el 15 de junio de 2016. Es un periodo que abarca cuatro meses y medio, que se traduce en 97 días laborales de dedicación.

La planificación se hace a partir del método de desarrollo definido en el ciclo de vida del proyecto, y se muestra en la Figura 36.



Task name	Start date	End date	Duration d...
Bachelor Project	01/02/16	15/06/16	97.00
Android IPC Monitor	01/02/16	15/06/16	97.00
Background	01/02/16	22/02/16	15.00
Identifiying Objetives	01/02/16	02/02/16	1.00
User Requirements	02/02/16	03/02/16	1.00
Related Works Study	03/02/16	12/02/16	5.00
Feasibility Study	12/02/16	15/02/16	1.00
System Architecture	15/02/16	17/02/16	2.00
Environment Setting	17/02/16	22/02/16	3.00
Incremental Development	22/02/16	18/05/16	62.00
Injector Module	22/02/16	03/03/16	8.00
Analysis	22/02/16	23/02/16	1.00
Design	23/02/16	24/02/16	1.00
Implementation	24/02/16	02/03/16	5.00
Test	02/03/16	03/03/16	1.00
Monitor Module	03/03/16	25/03/16	16.00
Analysis	03/03/16	08/03/16	3.00
Design	08/03/16	10/03/16	2.00
Implementation	10/03/16	24/03/16	10.00
Test	24/03/16	25/03/16	1.00
Commnunication Modul	25/03/16	19/04/16	17.00
Analysis	25/03/16	30/03/16	3.00
Design	30/03/16	04/04/16	3.00
Implementation	04/04/16	18/04/16	10.00
Test	18/04/16	19/04/16	1.00
Extraction Module	19/04/16	29/04/16	8.00
Analysis	19/04/16	20/04/16	1.00
Design	20/04/16	21/04/16	1.00
Implementation	21/04/16	28/04/16	5.00
Test	28/04/16	29/04/16	1.00
Data Analysis Module	29/04/16	17/05/16	12.00
Analysis	29/04/16	04/05/16	3.00
Design	04/05/16	09/05/16	3.00
Implementation	09/05/16	11/05/16	2.00
Test	11/05/16	17/05/16	4.00
Launcher Module	17/05/16	18/05/16	1.00
Documentation	18/05/16	15/06/16	20.00

Figura 36: Planificación temporal del proyecto (Gantt)

Capítulo 8: Presupuesto

8.1. Costes del Proyecto

En esta sección se desglosan los costes del proyecto.

La Tabla 72 contiene la información general del proyecto, donde se incluye la tasa de costes indirectos que se aplica en el sumatorio del coste final.

Información del Proyecto	
Título	Monitorización de Eventos en Android
Autor	Guillermo Izquierdo Moreno
Departamento	Departamento de Informática - UC3M
Fecha de Comienzo	1 de febrero de 2016
Duración	4 meses y 15 días
Tasa Indirecta	20%
Presupuesto Total	11.232,35 €

Tabla 72: Información del Proyecto

La Tabla 73 contiene el desglose de los costes derivado de la contratación de personal. Se construye a partir de las siguientes consideraciones:

- 1 Hombre mes = 131,25 horas.
- Máximo anual de dedicación: de 12 hombres-mes (1575 horas)
- Máximo anual para PDI de la Universidad Carlos III de Madrid: 8,8 hombres-mes (1155 horas)

Personal del Proyecto				
Nombre	**Categoría**	**Dedicación (hombre-mes)**	**Coste hombre-mes (€)**	**Coste (€)**
Sergio Pastrana Portillo	Doctor en Ingeniería	0,75	4.289,45	3.217,16
Guillermo Izquierdo Moreno	Graduado en Ingeniería	2,25	2.694,39	6.062,38
			Total	**9.279,53**

Tabla 73: Personal del Proyecto

La Tabla 74 describe el coste achacable a la adquisición y uso del equipamiento. El cálculo de la amortización se hace a partir de la siguiente fórmula:

Fórmula de cálculo de la Amortización: $\frac{A}{B} * C * D$

Donde:

- A = nº de meses desde la fecha de facturación en que el equipo es utilizado
- B = periodo de depreciación (60 meses)
- C = coste del equipo (sin IVA)
- D = % del uso que se dedica al proyecto (habitualmente 100%)

Equipos					
Descripción	**Coste (€)**	**Uso dedicado proyecto (%)**	**Dedicación (meses)**	**Periodo de depreciación (meses)**	**Coste imputable (€)**
Ordenador de sobremesa	750,00	100	4,50	60	56,25
Ordenador portátil	1.200,00	15	1,50	60	4,50
Dispositivo móvil	200,00	90	4,00	36	20,00
Impresora	300,00	2,5	0,10	60	0,01
				Total:	**80,76**

Tabla 74: Equipos

No ha habido subcontratación de tareas por lo que no se desglosa el capítulo.

La Tabla 75 muestra los costes directos imputables y no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros, etc.

Otros Costes Directos del Proyecto	
Descripción	**Coste imputable**
Licencia de software	0,00
Total	0,00

Tabla 75: Otros Costes Directos del Proyecto

El importe total para la realización del proyecto asciende a **11.232,35 € (once mil doscientos treinta y dos euros con treinta y cinco céntimos de euro)**.

Resumen de Costes	
Concepto	**Coste (€)**
Personal	9.279,53
Amortización	80,76
Costes de Funcionamiento	0,00
Cotes Indirectos	1.872,06
Total	11.232,35 €

Tabla 76: Resumen de Costes

8.2. Propuesta de Venta del Sistema

En esta sección se calcula un posible precio de venta que contemple los costes imputables al proyecto descritos en la sección anterior, un cierto margen económico por posibles riesgos y contingencias, el beneficio que se quiere obtener del proyecto y los impuestos sobre la venta de productos software. Este cálculo no se debe incluir en la oferta que se le envíe al cliente.

Propuesta de Venta			
Concepto	**Incremento (%)**	**Valor parcial (€)**	**Coste acumulado (€)**
Proyecto		11.232,35	11.232,35
Riesgos y Contingencias	15,00	1.684,85	12.917,21
Beneficio	30,00	3.875,16	16.792,37
Impuestos	21,00	3.526,40	20.318,77
		Total:	20.318,77

Tabla 77: Propuesta de Venta

El precio final asciende a **20.318,77 € (veinte mil trescientos dieciocho euros con setenta y siete céntimos de euro)**. Este es el precio que venta que se adjunta en la oferta al cliente, donde se incluye una cuota de mantenimiento anual de **3.358,47 € (tres mil trescientos cincuenta y ocho euros con cuarenta y siete céntimos de euro)**.

Mantenimiento Anual	
Porcentaje sobre el beneficio (%)	**Cuota de mantenimiento anual (€/año)**
20%	3.358,47
Total	3.358,47

Tabla 78: Mantenimiento Anual

Capítulo 9: Marco Regulatorio

9.1. Marco Regulatorio Aplicable al Software Desarrollado

9.1.1. Tipología del Software Según Su Licenciamiento

En el momento de publicación de este documento, el sistema no se encuentra bajo ninguna licencia software.

La pretensión es conseguir una licencia free software de tipo *X11* o *BSD 3-Clausule*, por las que se habilita la libre distribución de este software quedando el/los autores exentos de cualquier responsabilidad legal.

9.1.2. Software de Uso Legal

La utilización del software desarrollado no incumple la legalidad siempre que no se aplique sobre terceros sin su expresa autorización. De otra forma se estaría cometiendo un delito al vulnerar el artículo 197 del Código Penal Español aprobado por la Ley-Orgánica 10/1995, de 23 de noviembre.

9.1.3. Regulación Sobre la Propiedad Intelectual

El sistema desarrollado se atiene a la legislación y normativa relativa a la propiedad intelectual

Ámbito nacional.

- Real Decreto Legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual (LPI), regularizando, aclarando y armonizando las disposiciones vigentes en la materia.
- Ley 2/2011, de 4 de marzo, de Economía Sostenible (Vigente hasta el 02 de Octubre de 2016).

Ámbito europeo.

- Directiva 2001/29/CE del Parlamento Europeo y del Consejo, de 22 de mayo de 2001, relativa a la armonización de determinados aspectos de los derechos de autor y derechos afines a los derechos de autor en la Sociedad de la Información.
- Directiva 2009/24/CE del Parlamento Europeo y del Consejo, de 23 de abril de 2009, sobre la protección jurídica de los programas de ordenador

Ámbito internacional.

- Artículo 27 de la solemne Declaración Universal de Derechos Humanos, que contempla el derecho de toda persona a la protección de los intereses morales y materiales que se derivan de la autoría de las producciones científicas, literarias o artísticas.

9.1.4. Legislación Aplicable al Sistema de Información

El sistema desarrollado se atiene a la legislación y normativa relativa a sistemas de información.

Ámbito nacional.

- Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de carácter personal (LOPD).
- El Real Decreto 1720/2007, de 21 de diciembre, por el que se aprueba el Reglamento de desarrollo de la Ley Orgánica 15/1999, de 13 de diciembre, de protección de datos de carácter personal.

9.1.5. Normativas y Estándares

Normativas y estándares seguidos durante la construcción del sistema.

Ámbito internacional.

- IEEE STANDARD 830-1993 - IEEE Recommended Practice for Software Requirements Specifications.
- IEEE STANDARD 1016-1987 - IEEE Recommended Practice for Software Design Descriptions.
- ISO/IEC 12207:2008 establishes a common framework for software life cycle processes, with well-defined terminology, that can be referenced by the software industry.

9.2. Marco Regulatorio del Uso y Adquisición de Software Heredado

9.2.1. Tipología del Software Según Su Licenciamiento

El Sistema desarrollado incluye código del proyecto *ARM Inject*, publicado en un repositorio de Github. El proyecto es propiedad de Simone Margiratelli, el cual mantiene una licencia *BSD 3-Clause* sobre él con *copyright* de 2015.

El sistema desarrollado donde se incluye cumple las condiciones de uso y redistribución. Rl copyright es mantenido en el código fuente y reproducido en la documentación generada a la vez que no se promociona el sistema desarrollado citando al autor y se libra al mismo de toda responsabilidad legal.

9.2.2. Software de Uso Legal

La utilización del software adquirido no incumple la legalidad siempre que no se aplique sobre terceros sin su expresa autorización. De otra forma se estaría cometiendo un delito al vulnerar el artículo 197 del Código Penal Español aprobado por la Ley-Orgánica 10/1995, de 23 de noviembre.

9.2.3. Regulación Sobre la Propiedad Intelectual

El/Los autores reservan todos sus derechos sobre el proyecto *ARM Inject* sobre el que se aplica una licencia *BSD 3-Clause.* No está patentado.

Capítulo 10: Conclusiones y Futuros Trabajos

En este capítulo se detallan una serie de conclusiones extraídas durante el desarrollo del proyecto y la evaluación del sistema. Se valorará el cumplimiento de los objetivos destacando los aspectos positivos y negativos detectados durante el proceso.

Por último, se plantearán una serie de mejoras del sistema desarrollado, así como su evolución en futuros trabajos.

10.1. Conclusiones

Se puede concluir que se ha conseguido desarrollar una herramienta ligera, que cumple con el objetivo inicial. Se consiguen extraer del dispositivo los datos relativos a la comunicación entre procesos de forma efectiva. Además, esta información se enriquece con los datos extraídos de llamadas al sistema regulares que pueden considerarse precedentes o consecuentes de la comunicación entre procesos.

Se considera muy alentador que la herramienta tenga una utilidad real. A partir de la información capturada, es posible reconstruir eventos utilizando métodos estadísticos para revelar ciertos patrones de comportamiento.

Un aspecto a tener en consideración es que, pese a la enorme cantidad de información que está filtrando, no hay pérdidas de paquetes siempre que se trabaje sobre una red no saturada, como puede ser una red privada. Esto es un dato positivo porque revela que el diseño e implementación de arquitectura garantiza un nivel de rendimiento adecuado para esta tarea. Además, es una herramienta robusta. Puede estar funcionando durante largas sesiones de recolección de datos.

Como se ha podido apreciar en la sección 6.1, existe una limitación del sistema cuando trabaja sobre redes saturadas, pues se han apreciado pérdidas de paquetes en esas condiciones. Las redes públicas no son las más apropiadas para realizar experimentos que requieren cierto rendimiento, pues en ellas suelen estar conectados simultáneamente muchos terminales. La recomendación es que se trabaje sobre una red privada.

Uno de los aspectos negativos de este sistema es la necesidad de escalar privilegios para poder ejecutar su funcionalidad. Los permisos de superusuario no están habilitados en un sistema operativo Android original, pero es posible habilitarlos mediante la ejecución de herramientas específicas. Así se consigue lo que coloquialmente se denomina un móvil *rooteado*.

Durante el desarrollo de este proyecto se ha manifestado riesgo que puede llegar a suponer el hecho de habilitar privilegios de superusuario en el dispositivo móvil de un usuario común. Se pueden alterar con facilidad las políticas de seguridad impuestas por SELinux, acceder a todo el espacio de usuario, cambiar permisos sobre ficheros y directorios e invocar las funcionalidades del *Kernel* a través de llamadas al sistema con total libertad. Un ejemplo muy básico del nivel de riesgo al que se expone un usuario es, por ejemplo, que se permite acceder a un fichero del sistema donde se almacenan en claro las contraseñas de las redes de área local protegidas por WPA a las que el dispositivo se puede conectar (*wpa_supplicant.conf*).

Hay que tener en cuenta que no solo se beneficia el usuario de la escalada de privilegios, sino que también se beneficia el malware, el cual podría actuar sin ninguna limitación en el dispositivo en caso de infección.

El propio sistema desarrollado, utilizado de forma malintencionada, podría pasar por una pieza de malware. La capacidad para interceptar las llamadas al sistema que invoca un proceso, entre las que se incluyen las que implican comunicación entre procesos, manifiesta el nivel de introspección en la plataforma que posibilitan estas herramientas. Normalmente, los mecanismos de cifrado están implementados en hardware o a nivel de *Kernel*, por lo que los procesos confían la confidencialidad de los datos al *Kernel*. Las llamadas al sistema son interceptadas antes de que pase a ejecutarse su rutina en el *Kernel*, por lo que se pueden obtener datos confidenciales en claro. Una forma de solucionar esto consistiría en implementar mecanismos de cifrado software, pero esta medida afectaría muy negativamente al rendimiento del sistema.

Otro aspecto a mejorar es la posibilidad de que la monitorización de las llamadas al sistema por parte del proceso no se absolutamente transparente al propio proceso. Un proceso puede comprobar si las llamadas al sistema que invoca han sido redirigidas, lo que haría al sistema detectable en caso de analizar malware. En todo caso, se estaría hablando de malware sofisticado que hoy en día es menos frecuente para Android, si bien no hay que sobrestimar la evolución del mismo.

Una de las tareas que no se ha realizado es el unmarshalling efectivo de los paquetes de datos interceptados en la comunicación entre procesos. Este no es un procedimiento trivial, por lo que se ha decidido que de esos paquetes se van a almacenar crudo ya que se considera que conociendo la función y el código del método llamado remotamente aporta suficiente información para clasificar comportamientos.

La última conclusión que se puede extraer, es que el autor considera que ha adquirido un conocimiento profundo en el funcionamiento completo del sistema operativo Android a varios niveles de abstracción, además de haber adquirido las capacidades de programar tanto aplicaciones en Java como componentes nativos. Esto permitirá continuar con tareas de investigación en Android en un futuro.

10.2. Trabajos Futuros

La detección de defectos o posibles mejoras en el sistema junto con la propuesta para enmendarlos esos defectos y la sugerencia de técnicas novedosas o más eficientes forman parte del proceso de desarrollo del sistema. Lo que se pretende con esta sección es evitar el estancamiento del sistema tal y como fue desarrollado originariamente y dar paso a una evolución.

El planteamiento del análisis de comportamientos de aplicaciones en Android se ha materializado a través de diferentes técnicas, no hay una solución única, por lo que la tendencia es la combinación de las diferentes técnicas para poder refinar la herramienta.

En la sección de conclusiones se han comentado ciertos aspectos negativos que se podrían mejorar.

- La automatización de la tarea de monitorización para analizar aplicaciones a gran escala. Se podría adaptar el sistema *Baredroid* [11] para esto.
- Realizar un análisis más en profundidad de los eventos del Binder, mediante el *unmarshalling*, al igual que lo realizado en *CopperDroid* [1].
- Mejorar las técnicas de análisis, incorporando mecanismos de minado de datos y clasificación por patrones para detectar comportamientos más precisos, no basados en métodos estadísticos.

Anexo 1: Implementación ARMINJECT

Este anexo incluye la descripción de la implementación del módulo inyector heredado del proyecto ARM Inject, propiedad de Simone Margaritelli [9]. Los pasos que se siguen en la implementación para asociarse al proceso objetivo se explican a continuación.

En primer lugar se ejecuta una llamada *ptrace()* sobre el proceso objetivo *PID* con el código de operaión *PTRACE_ATTACH*, el cual se asocia al proceso objetivo (*tracee*) y le manda una señal de parada de ejecución *SIGSTOP*, acción no instantánea, por lo que tiene que esperar por él.

Para forzar al proceso *tracee* a ejecutar llamadas remotas hay que resolver la dirección de dichas llamadas en su espacio de memoria virtual.

Todo proceso que se ejecuta sobre el Kernel de Linux tiene mapeado el *linker* dinámico */system/bin/*linker, donde se encuentran las funciones de manejo de biblioteca dinámica, *dlopen*, *dlsym* y *dlerror,* y la biblioteca estándar de C, */system/lib/libc.so*, llamada *Bionic* en la versión para Android, donde se encuentran las llamadas básicas en un entorno de programación en C, en su espacio de memoria virtual.

Los símbolo dentro de un objeto compartido van a tener el mismo desplazamiento en memoria desde la dirección virtual base del objeto compartido en la memoria virtual del proceso, la cual se puede conocer, pues se encuentra en */proc/<PID>/maps*.

Las direcciones virtuales remota de los símbolos *dlopen*, *dlsym, dlerror, calloc* y *free* se resuelven aplicando la diferencia del desplazamiento entre la dirección del objeto compartido en el proceso *tracer* (local) y el *tracee* (remoto).

remote_address = remote_base + (local_addrees – local_base)

Resueltos los símbolos remotos, hay que forzar al proceso *tracee* a ejecutar las llamadas a esos símbolos alterando sus registros. Para ello se ejecuta la llamada *ptrace*, con el código de operación *PTRACE_GETREGS* para salvar los registros que tenía cargados el proceso *tracee*, el cual sigue parado desde que se ha dado la asociación. Se cargan los argumentos de la función remota a invocar entre los registros R0 - R3 y en el *stack* de forma ordenada. En el registro LR, dirección de retorno, se carga un 0, para que resulte en una violación de segmento, *SIGSEGV*, la cual se captura. En el registro PC, dirección de siguiente operación, se carga la dirección de la función remota que se quiera ejecutar. Por último, se vuelve a ejecutar la llamada *ptrace* con el código de operación *PTRACE_SETREGS* para cargar los registros en el proceso *tracee*, y se ejecuta la llamada *ptrace*, con el código de operación *PTRACE_CONT* para que continúe la ejecución del proceso, se espera por él hasta que termine de ejecutar la rutina, cuyo valor de retorno se

encuentra en el registro R0, y se reestablecen los registros con los que se ha encontrado al proceso.

Este método de invocación forzosa de símbolos remotos se aplica para que el proceso remoto reserve memoria en su espacio de direcciones invocando a *calloc*. A continuación, invocando *ptrace* con código de operación *PTRACE_POKETEXT*, se escribe el *linker name* de la biblioteca que se quiere cargar en el proceso remoto en la dirección de memoria remota que ha retornado la reserva, de forma que esté alineado en memoria. Por último, fuerza al proceso *tracee* a ejecutar *dlopen* sobre la dirección de su espacio de memoria virtual donde se encuentra el *linker name* de la biblioteca a inyectar.

Anexo 2: Abstract

Introduction

The analysis of Android applications is a field that has been a hot topic area of research during the last years. The main reason is that Android is the most popular mobile operating system in the market, therefore, it is the system being targeted by most samples of malware. The raising of Android malware is partially due to its *open source* project, which means that its code is exposed to be investigated seeking for vulnerabilities.

This situation has also been encouraged by Android installation policy. The system allows installation of signed applications, but accepts the self-signed by any author, it does not require a certification authority. In addition, application integrity and updates remains on the honesty of the author. This has led it to become the mobile platform where more pieces of malware are distributed, for ease of entry for Trojans.

Research into this is based on a clear premise: it is necessary to perform a dynamic analysis of applications that can complement static analysis. Malware is often included in obfuscated codes unworkable in a single static analysis. In addition, forensic evidences that the malware leaves after being executed in a system often does not provide a significant amount of data that allows to understand its behaviour. It is necessary to collect information during the execution of malware, as this is the only way to retrieve values stored in memory, communication between processes, network connections, etc.

Alongside this broad area of research, many dynamic analysis tools of malware behaviour have been developed in different forms. Some are based on emulated or virtualized environments such as *CopperDroid*, *DroidScope* or *Google Bouncer*. Other proposals are based on real platforms, as *BareDroid*, but the goal to achieve is the same for all of them: to get a transparent and complete monitoring of the applications.

Evasive malware can fingerprint such analysis systems, and, as a result, they prevent the execution of any malicious activities. Most of the fingerprinting techniques exploit the fact that dynamic analysis systems are based on virtualized or emulated environments, which can be detected by several known methods. [10]

Despite the differences in its architecture, the common analysis approach relies on intercepting the regular system calls and the inter-communication process in Android, which is performed mostly through a tool named Binder, which works on the operating system kernel.

The goal of this project is to build a tool that monitors events generated during the execution of an Android application, so as to detect patterns of malicious activities.

The analysis tools seeking for malware behaviour are frequently built on the top of emulators can be detected by the malware samples themselves, by testing a number of features that reveal discrepancies between an emulated platform and an unmodified operating system installed on a real device (*bare-metal*) .

One challenge faced by a malware analysis tool running on a *bare-metal* system is the extraction of data generated from the mobile device. The extraction is impracticable without introducing any analysis component on the hosting system, thus it makes the system detectable which violates the transparency requirement.

Android Overview

Android is an operating system based on a simplified Linux standard kernel but enhanced with specific extensions for mobile needs.

The whole system implementation relies programming languages from different nature: Java, C++, C and Assembler. These languages can be found along the system in different abstraction layers corresponding to their properties/domain, e.g. Java is used in user APPs, C++ is used in native applications and Android libraries, and Assembler and mainly C in the kernel.

Since Android OS is user-interaction oriented, most of the runnable programs are distributed as Android Packages Archive (APKs) which are considered to be self-contained applications composed by one or more components where user interfaces are usually included.

Those applications are executed in a controlled environment. The *Dalvik Virtual Machine* (DVM) is an isolated runtime environment where Java programmed applications are processed into its own bytecodes and executed, similarly to Java Virtual Machine (JVM). It is register based for performance reasons and it’s highly compatible with ARM hardware.

There is a difference between programs compiled for the virtual machine and programs compiled to run on a specific computation platform, like Intel x86 or ARM. Those compiled for a specific platform are called native and can be executed directly, unlike Java, which is executed in a virtual machine.

Each application is executed as an instance of the DVM in its own user-space process isolated in a sandbox, where is applied a minimum privilege policy for security reasons. An application can only access system or third-party components if it explicitly requested the corresponding permissions in its manifest file [1]. The way for an application to access low-level OS mechanisms, as kernel calls, is through the *Java Native Interface*, which allow Java to execute compiled code from libraries written in other languages (e.g. C++) making use of a well-defined APIs [2].

In addition, every installed application has its unique user ID and group ID and may only manipulate data from its own directory. All apps must be signed by the author, and, only in case of be signed by the same author, can run with the same UID and GID, which is a problem because they can communicate freely and their permissions accumulate [2].

There is a set of Android application components (e.g. activities, services, etc.) that performs different tasks and can be invoked by the operating system or directly by the user. When different components have to exchange data or perform actions in other components, the inter-component communication takes place making use of *intents*, which are asynchronous messages that contains an URI that uniquely identifies an application component and an action that identifies the operation to be executed. It is not necessary that the component is activated from the application itself, it may be the result of an inter-process communication [2].

Android Binder

Binder is an Android-specific inter-process communication mechanism, and remote method invocation system. An Android process can call a routine in another Android process, using Binder to identify the method to invoke and pass the arguments between processes [2].

Commonly, Android system utilities are offered to the user applications as remote services that can be invoked as if they were local.

It is a characteristic of AIDL thus settled in a higher level, that an application does not need to know if a service is working in a server process or in the local process. The application concept of Android facilitates to run a service either in an own process or in the activity process. Therefore, it is possible to export services between applications.

A service that offers a utility is referred by an instance of Binder. Each Binder can act as shared token as it is uniquely identifiable. Even when a Binder is not published via the service manager, its identification is possible due to the *token*. Consequently a Binder may be used as a security access token. The token can also be shared across multiple processes.

A caller process can be identified by its callee by UID and PID. In addition, a heap of shared memory can be shared using Binder.

The Binder ensure a well object oriented IPC.

Binder Transactions

Each packet sent between processes is called Binder Transaction. It contains a binder token that identifies the destination object, a code of the action to perform, a raw data buffer, and sender PID/UID.

An *ioctl* system call must be executed in order to launch a Binder transaction to a target service, its arguments are the file descriptor /dev/binder, the Transaction Code and a transaction structure. When invoked, the Binder Driver resolves the destination object, copies the data to the server's address space and wakes up a waiting thread in the server process to handle the request. The server follow the same steps to send the response to the client.

When the request code sent to the Binder Driver via *ioctl* call is *BINDER_WRITE_READ*, the driver will deal with IPC data. This command is the basis for Binder IPC transactions. When this code appears, the structure of the buffer pointed by the third argument is well-known and can be found in *binder.h*. (See Figura 3)

Binder IPC relies on a synchronous request-reply mechanism. This fact implies that a requesting process will be blocked waiting for the response unless the asynchronous way is specified with the *TF_ONE_WAY* flag. Generally, a transaction implies two messages: a transaction request and its reply. The one way communication of the Binder is limited to internal features as the *death notification*. [2]

Request and reply data will be contained in the *struct binder_write_read* after the *ioctl* call. The request data is pointed by *write buffer* and the reply data is pointed by *read_buffer*. Both data buffers are prefixed by a four *bytes* code that identifies the type of request and reply. All possible codes are listed in *enum binder_driver_command_protocol*.

If the code at the beginning of *write_buffer* is *BC_TRANSACTION*, then the driver will deal with an IPC transaction, and this buffer can be parsed to a *struct binder_transaction_data*, also declared in *binder.h*. (See Figura 4)

The transaction structure contains:

- *target*: identifies the destination Binder node making the distinction whether it is invoking an object in the memory space of a remote process or in the current process.
- *data.ptr.buffer:* points to a serialized buffer where each entry corresponds to a command and its invocation arguments. That command usually identifies the AIDL of a system service, and the arguments are passed to the method remotely invoked.
- *code:* identifies the method invoked. It is resolved by the service's stub, that unmarshalls the parceled arguments and invoke the corresponding method of the AIDL. Those methods are generally ordered in a way that its position into the AIDL matches with its code starting at *IBinder::FIRST_CALL_TRANSACTION*, defined as 1.
- *sender_pid:* consist in the requesting process *PID*, and it's used to check if the process has permissions to execute the requested service.

This structure provides a meaningful amount of data that can be used with different purposes: analyzing IPC behaviors, filtering relevant data, calculate system statistics, etc.

Objectives

This project proposes a system designed to minimize the probability of being detected. The internal analysis component is limited to the interception and the extraction of the data. The interception is considered a critical task, where the performance and the transparency are crucial, while the data extraction could be delegated to other process.

In order to collect the data, the proposed tool intercept system calls invoked by the target process. These system calls include one which involves inter-process communication (IPC) through the Binder Kernel Driver, i.e. the *ioctl*. The analysis of this system call require special attention, as they consume the essential mechanism of Android IPC.

The system is designed for being executed on an Android OS installed on a real device, not an emulated platform. This guarantees precision and a higher degree of transparency during the data collection.

Monitoring is performed at user space due to portability reasons, without the need to modify the operating system kernel. The only requirement for the device is that the operating system has to allow root privilege escalation and that the security policy of SELinux acquire the most permissive level.

A native level process carries out the intercepting task, which is responsible for intercepting the most significant system calls during the execution of an app in Android. The extracting task is performed by another process for transparency, functionality and performance reasons. This process sends the data over the network to a server for analysis, where data is interpreted and stored for further analysis.

System overview

The monitoring is implemented as part of a dynamic library, which is implemented at the Android native level and loaded in the target process to intercept system calls. Since the process is already running, an injector module enforces the dynamic library load. The functionality of the library is not responsible for sending the data intercepted from the device to the server. In order to do this, the extraction is controlled by another process running on an Android application, being part of the system. This application contains the logic required to send the data via network. Communication between these two processes is established through shared memory to avoid using the same IPC mechanism being monitored (i.e., the Binder) and benefit from the excellent performance of communication through shared memory. Once received the data, the server running in an external PC handles the storage and analysis of the data.

Architecture

Injector Module

This module induces the target process to load an arbitrary dynamic library into its memory space. It is achieved using the *ptrace* system call to get attached to the target process and to modify its registers to force the loading. For the implementation of this module, we have adopted an open-source project and adapted it to the requirements of our tool.

Monitor Module

It is contained in the dynamic library loaded by the target process. This is where the functionality of system calls interception is actually implemented. The technique used is the "*PLT patching*" or *hooking*, which replaces the target process addresses where the original system calls are mapped with the addresses of our hooking routines. These hooking routines are in charge of collecting the system call arguments and sending them to the communication module.

The data to be exfiltrated (e.g. arguments to calls) are packaged as a *byte* chain, whose format depends on the type of system call. Moreover, these packets are sent to the extraction process, which is running in an Android application, through the mechanism that habilitates the Communication Module.

Communication Module

This module is in charge of establishing the communication between the Monitor Module and the Extracting Module, thus, the data intercepted is packaged and sent to the extracting process.

The mechanism used for this purpose is the shared file mapping in memory.

It is composed by a client process, which is included within the Monitor Module in the dynamic library, and a server process, which is executed as part of the Extracting Module in the extraction application.

The client side opens a file in the Android shared memory allocator, *ashmem*, and it maps a region of the file in memory. The client side has to share the mapped file descriptor, so the process running in the server side is able to map it and, consequently, to notify changes in memory immediately.

The file descriptor sharing is done using the Android Binder IPC. The server side registers a native service in the system that offers a mapping function. This function maps the file descriptor received in memory. The client side has to consume the remote function to share the file

descriptor. It is important to remark that this Binder interaction occurs before the beginning of the monitoring process, and, therefore, it does not alter the normal analysis of events since it is not recorded in the extracted data from the external server.

The client side is implemented in C/C++, as a part of the injected library. The server side has both Java and native (C/C++) components, which communicates through the Java Native Interface (JNI).

Extracting Module

This module sends the data received by the Communication Module to the external analysis server. It is implemented in the extraction application (packaged as an Android APK) and it is entirely implemented in Java.

The communication mechanism used between the Communication Module and the Extracting Module is a double reference to a *Handler* object instanced in each process. This object is attached to the thread and its message queue. It allows both processes to send and receive messages. The requests sent are *Message* objects, which are managed by Android Binder IPC.

This module is executed as the main activity of the application, so it is not supposed to invoke blocking calls. The network activity is performed by an asynchronous task, and packets are sent through UDP to the external server.

Data Analysis Module

This module is responsible for the analysis and processing of the data obtained during the execution of the targeted process, in order to detect behavioural patterns. It is the only module one that resides entirely on the remote server.

In its current version, it generates statistics from the most commonly used system calls and IPC events collected from the monitored application, both in absolute terms and in the call distribution along the time, and provide a visual chart with such information. Using this analysis, it is possible for a human operator to understand what the dynamic used by the applications and their interaction with the Kernel and other processes are.

The modular nature of the design proposed in this project allows the analysis module to be updated in the future to enrich the data analysis, for example using data mining techniques and pattern matching to recognize patterns and help to detect malicious applications.

Data Analysis Module

This module consists in an activation script. It is in charge of start the extraction application as well as the Injector Module so the monitoring can start. In addition, it sent metadata of the monitored process to the remote server to make easier the data interpretation.

Experimentation

To show the usefulness of the developed tool, there have been accomplish two experiments as a proof of concept. In these experiments, we first analyze the activity of the Google Chrome application and integrated calculator Android.

The Google Chrome Tests consist in establishing a connection to a web page, and to download a PDF file, which will be store in the device. In this test it is possible to notify the high amount of system call invocations relate to the reception of data through network (*recvfrom*), and writing a file (*write*) with the downloaded data. Such behavior was expected sin the nature of the behavior.

A peak of *write* and *recvfrom* system calls may be appreciated along the monitoring time, which corresponds to the period when the downloading took place.

The calculator test is so simple. It consist basically in monitoring an addition operation between two numbers. The objective of that short experiment is to show the enormous amount of system calls that can generate a simple operation in an Android environment. It shows more than 1200 system call events, where numerous *recvfrom* system calls are included.

At first glance, it seems strange that network interface operations are involved in a simple adding, but analyzing the arguments of the system call, it can be deduced that the application is using a *socket* communication in a local way, which means that it may be consuming resources of the operating system.

This gives an idea of how complicated it can be to detect behaviours in a dynamic analysis, because an application may perform operations that seems malicious, but actually they are not.

Conclusions

It has been achieved the development of a soft tool, that accomplish the initial objective. The extraction of data from the device related to inter-process communication is effectively extracted. Furthermore, this information is enriched with data extracted from the regular system calls, which may be considered as precedents and consequents of the IPC.

It is considered very encouraging that the tool has a real utility. From the information captured, it is possible to reconstruct events using statistical methods to reveal certain patterns of behaviour.

An aspect worth to mention is that, despite the enormous amount of information that is being leaked, no packet loss took place whenever the system works on an unsaturated network, such as a private network. This is a positive point, because it reveals that the design and the implementation of the architecture ensures an adequate level of performance for this task. In addition, it is a robust tool. It may be running during long sessions of data collection.

There has been noticed a limitation of the system when working on saturated networks, packet loss has been appreciated under these conditions. Public networks are not appropriate to perform experiments that require certain performance, because there are often many terminals connected simultaneously. The recommendation is to work on a private network.

One of the negative aspects of this system is the need to escalate privileges to run their functionality. Superuser permissions are not enabled in an original Android operating system, but it is possible to enable them by running specific tools. This is colloquially called "*rooted mobile"*.

During the development of this project it has been demonstrated the risk that may suppose enabling superuser privileges on the mobile device of a common user. It is possible to easily alter security policies imposed by SELinux, access to all user space, change permission on files and directories and freely invoke the Kernel functionality through system calls. The escalation of privileges benefits, not only the user but also the malware, which could act without any limitation on the device in case of infection.

The developed system itself, used in a malicious way, could accomplish malicious activities characteristic of malware. The capacity to intercept system calls invoked by a process, among those involving inter-process communication are included, manifests the level of introspection on the platform that enables these tools.

Another aspect is the possibility to improve the transparency in system calls monitoring. A process can check whether the system calls have been redirected, which would make the system detectable in case of analyzing malware.

A task that has not been done is the unmarshalling of the data packets intercepted communication between processes. This is not a trivial procedure, so it was decided that these packages are stored as raw data.

The final conclusion to be drawn is that the author believes he has acquired a deep knowledge in the full functionality of the Android operating system at various levels of abstraction, besides having acquired programming skills both in Java applications and native components. This will allow the author to continue conducting research in the future.

Future Works

The detection of defects and possible improvements in the system alongside the proposal to fix them and the suggestion of novel or more efficient techniques are part of the system developing. The aim of this section is to avoid the impasse of the system just like It was originally built and, therefore, to promote an evolution.

The approach the behavioral analysis of Android applications has been materialized through different techniques, not a unique one. Thus, the tendency is to combine such techniques in order to refine the tool.

In the conclusions, certain negative aspects have been mentioned with the objective of them to improve:

- The automatization of the monitoring task to make high-level analysis. The *Baredroid* [11] system could be adapted for this purpose.
- To perform an in-depth analysis of Binder events through an effective unmmarshalling, as well as it was accomplished in *CopperDroid* [1].
- To improve the analysis techniques, including data mining mechanisms and pattern classification to detect more accurate behaviours, not based in statistical methods.

Bibliografía y Recursos

[1] S. J. K. A. F. a. L. C. Kimberly Tam, "CopperDroid: Automatic Reconstruction of Android Malware Behaviors," 2015.

[2] J. S. D. B. Thorsten Schreiber, «Android Binder Android Interprocess Communication,» 2011.

[3] «Security-Enhanced Linux in Android,» [En línea]. Available: https://source.android.com/security/selinux/.

[4] «Android Studio 2 Development Essentials,» [En línea]. Available: http://www.techotopia.com/index.php/An_Overview_of_the_Android_Architecture.

[5] A. Developers, «Android Fundamentals,» [En línea]. Available: https://developer.android.com/guide/components/fundamentals.html#Resources.

[6] A. Gargenta, «Deep Dive into Android IPC/Binder Framework,» 2013.

[7] I. R. Nitay Artenstein, «Man in the Binder: He Who Controls IPC, Controls the Droid,» 2014.

[8] «Shared Memory Introduction,» [En línea]. Available: http://www.kohala.com/start/unpv22e/unpv22e.chap12.pdf.

[9] S. M. (evilsocket), «Dynamically Inject a Shared Library Into a Running Process on Android/ARM,» [En línea]. Available: https://www.evilsocket.net/2015/05/01/dynamically-inject-a-shared-library-into-a-running-process-on-androidarm/.

[10] G. V. C. K. Dhilung Kirat, «BareCloud: Bare-metal Analysis-based Evasive Malware Detection,» 2014.

[11] Y. F. A. B. L. I. J. C. D. K. C. K. G. V. Simone Mutti, «BareDroid: Large-Scale Analysis of Android Apps on Real Devices,» 2015.

[12] S. Y. W. T. Z. X. Z. H. PENG Guojun, «Research on Android Malware Detection and Interception Based on Behavior Monitoring,» 2012.

[13] H. Y. Lok Kwong Yan, «DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis,» 2012.

[14] «Apriorit - Redirecting functions in shared ELF libraries,» 2010. [En línea]. Available: https://www.apriorit.com/dev-blog/181-elf-hook.

[15] S. S. a. R. Craig, «Security Enhanced (SE) Android: Bringing Flexible MAC to Android,» 2013.

[16] «SELinux Wiki - Access Vector Rules,» 2009. [En línea]. Available: http://selinuxproject.org/page/AVCRules.

[17] «Linux Man Pages - strace,» [En línea]. Available: http://linux.die.net/man/1/strace.

[18] «Linux Man Pages,» [En línea]. Available: http://linux.die.net/man/.

[19] I. Object Management Group, «OMG Unified Modeling Language™. Version 2.5.,» 2015. [En línea]. Available: http://www.omg.org/spec/UML/2.5.

[20] A. Petrov, «Andrey's blog - Android hacking: hooking system functions used by Dalvik,» 2013. [En línea]. Available: http://shadowwhowalks.blogspot.com.es/2013/01/android-hacking-hooking-system.html.