Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Modificación del control de recursos en el sistema operativo Android

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Ayelén Chavez

Joaquín Rinaudo

Codirectores:

Santiago Ceria

Juan Heguiabehere

Gerardo Richarte

Buenos Aires, 2013

Índice general

Modificación del control de recursos en el sistema operativo Android

Android es un sistema operativo basado en *Linux* con millones de usuarios en todo el mundo. Su esquema de seguridad para proteger al usuario de aplicaciones abusivas es de todo o nada. Al momento de querer instalar una aplicación, el usuario debe aceptar todos los permisos que la misma requiere en bloque, sin poder modificarlos una vez que fueron concedidos.
A falta de una mejor implementación, desarrollamos una aplicación que permite al usuario un control más granular y flexible sobre el acceso a aquellos recursos que consideramos de mayor importancia para el usuario desde el punto de vista de su privacidad y seguridad.
Cuando una aplicación solicita acceso a un recurso, nuestro enfoque permite al usuario no sólo devolver datos falsos o anónimos, sino también seleccionar un subconjunto de los datos reales, para que pueda seguir usando las aplicaciones pero de manera controlada. De esta forma, un usuario puede defenderse de posibles abusos de confianza que se toman algunas aplicaciones y que el esquema de seguridad de Android no aborda. En otros trabajos relacionados donde sólo se puede devolver datos falsos, su control perjudica la usabilidad cuando se trata de un recurso clave para la aplicación. Por ejemplo, podríamos considerar sin sentido usar una aplicación como *WhatsApp* con una agenda de contactos vacía.
Logramos controlar el acceso a contactos, datos que permiten identificar unívocamente a un dispositivo y a su dueño como ser el identificador del equipo, identificador del subscriptor, tarjeta sim y número de teléfono, información de red del dispositivo, como ser IP, MAC, la SSID y BSSID del punto de acceso al que está conectado, redes configuradas y escaneadas y, finalmente, controlamos toda forma en la que una aplicación puede saber la ubicación del dispositivo, no sólo usando el gps sino también las torres de celulares y redes inalámbricas.

Palabras claves: Android, Seguridad, Recursos, Permisos, Privacidad, Abuso de Confianza, Control de Acceso, Monitor de Referencia, ASA.

Modification to the Android operating system's resource control

Android, an operating system based on the *Linux* kernel, has millions of users around the world. Its security scheme to protect a user from abusive applications is all or nothing. When a user wants to install an application, she must accept every permission it's requiring as a whole, without the chance to change them once they were granted.
For lack of a better implementation, we developed an application that allows a more granular access control to the resources we consider of main importance from the user's privacy and security point of view.
When an application asks to access a resource, our approach allows the user not just to retrieve false or anonymous information, but to select a subset of the real data, so she can keep on using the applications in a controlled manner. This way, a user can defend herself from a possible abuse of trust that some applications may attempt. The Android security architecture doesn't address this kind of trust problems. In other related works where the user can only retrieve fake data, access control affects the usability when controlling a key resource for the application. For example, the user clearly did not intend using *WhatsApp* with an empty contacts list.
We've accomplished control access to contacts, data that allows a device and its owner to be unambiguously identified such as its id, subscriber id, sim card and phone number, device's network information such as its IP, MAC, the SSID and BSSID of the access point it is connected to, the configured and scanned networks and, finally, we control every way an application can know the device's location, not only using the gps but also the cell towers and configured wireless networks.

Keywords: Android, Security, Resources, Permissions, Privacy, Abuse of Trust, Access Control, Reference Monitor, ASA.

Agradecimientos

A nuestros viejos, nuestros directores y a la UBA

Capítulo 1

Introducción

1.1. Qué es Android

Android es un sistema operativo para dispositivos móviles basado en Linux desarrollado por *Google*. El mismo provee un repositorio de aplicaciones, denominado *Play Store*, que contiene una enorme cantidad de aplicaciones, no sólo pagas, sino también gratuitas y de diversa procedencia. Datos oficiales revelan que Android controla el 80 % del mercado de teléfonos inteligentes a nivel mundial. Para dar un número concreto, en el 3er cuarto de 2013, se vendieron 187.3 millones de dispositivos con el sistema operativo Android entre ellos celulares, tablets, TVs, etc.

1.2. Motivación

1.2.1. Abuso de confianza

Con abuso de confianza nos referimos al uso o aprovechamiento excesivo o indebido de información de un usuario de Android por parte del fabricante del dispositivo, la compañía de telefonía o los desarrolladores de aplicaciones instaladas en el mismo.

Cuando compramos un nuevo dispositivo o cuando elegimos una compañía de telefonía, depositamos confianza en ambos, así como en el conjunto de aplicaciones preinstaladas que pertenecen a alguno de estos. También, al momento de instalar una aplicación, depositamos confianza en quienes la desarrollaron.

En nuestro equipo dejamos mucha información personal, sensible y privada, por lo que el hecho de que alguno de estos sujetos usen el acceso que tienen a nuestra información para beneficiarse de alguna forma, debe ser controlado.

Un ejemplo típico es el de una aplicación que tiene acceso a la agenda de contactos de un usuario y los publica para que sean utilizados en las listas de ventas de distintas empresas que realizan venta telefónica. Otro caso de abuso de confianza es el uso de datos del equipo que permiten identificar a una persona para hacer un análisis de mercado, incluyendo los gustos del usuario, actividades diarias, hobbies, etcétera y luego usar estos datos como estrategia de venta sin previo consentimiento del usuario.

Un caso puntual es el detallado en el artículo [**11**] donde una linterna copiaba la ubicación y el código de identificación del dispositivo para luego enviarlos a empresas comercializadoras, a pesar de que sus políticas de privacidad decían que no lo haría.

Creemos que el esquema de seguridad de Android no es lo suficientemente completo para controlar o evitar estos problemas de abuso de confianza. Por ello, una de las metas principales de este trabajo es darle una herramienta al usuario que le permita, basado en su propio discernimiento, defenderse de estos abusos.

1.2.2. Sistema de permisos de Android

La administración de permisos requeridos por una aplicación realizada por Android es muy sencilla: el o los desarrolladores declaran los permisos que necesita una aplicación en un archivo llamado manifiesto. Dicho archivo es configurable y es incluido en el instalador para ser mostrado al usuario durante el proceso de instalación. Luego, se requiere que el usuario acepte el manifiesto como paso previo a la instalación. Una vez que la instalación se realiza exitosamente, el sistema operativo sólo permite que la aplicación acceda a los recursos para los cuales tiene permisos según fueron concedidos por el usuario durante la instalación, pero no existen mecanismos para modificar dichos permisos más adelante. Es decir, el otorgamiento de dichos permisos es estático.

En la figura 1 se muestra la lista de permisos requerida por *Facebook*, declarando algunos permisos que podrían llevar a un abuso de confianza como llamadas directas a otros dispositivos o permiso para leer los mensajes de texto, ya que no son estrictamente necesarios para el funcionamiento de la aplicación.

Una aplicación puede pedir diversos permisos, algunos de los cuales permiten el acceso a nuestra información personal, como por ejemplo acceder a nuestra agenda de contactos, a nuestro historial de llamadas, etcétera. Otros permisos permiten acceder a recursos como realizar llamadas con costo, conectarse a internet o la cámara.

Otro detalle no menor a tener en cuenta es que los desarrolladores de aplicaciones Android no están orientados a ser minimalistas al momento de definir qué permisos utilizará su aplicación. Además, al actualizar una aplicación que ya tenemos en nuestro dispositivo, ésta podría pedir permisos adicionales, siendo esto algo que puede pasar desapercibido por el usuario promedio. Hay aplicaciones que no deberían usar casi ningún recurso de nuestro dispositivo, como ser una linterna, pero resulta que la misma se conecta a internet y tiene acceso a nuestra cámara de fotos o a nuestro log de llamadas telefónicas.

1.2.3. Datos sensibles y privados

Hoy en día un usuario de un dispositivo móvil usa el mismo para realizar tareas muy variadas de su ámbito personal y laboral; jugar juegos, escuchar

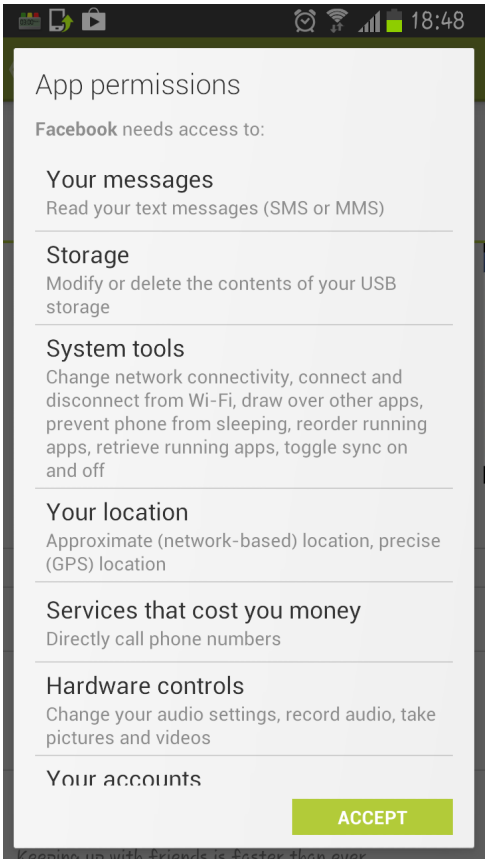


FIGURA 1. Listado de algunos de los permisos de la aplicación *Facebook*

música, sacar fotos, grabar videos, navegar por internet, acceder a sus cuentas bancarias, enviar mensajes de texto, emails, realizar llamadas, conocer los caminos disponibles para ir de un lugar a otro, además de poder bajarse todo tipo de aplicaciones que le permitan al usuario delegar sus actividades a su equipo. Desde una simple linterna que usa el flash de nuestro celular hasta la aplicación de la red social de moda que tiene acceso a todo lo que hacemos, incluyendo dónde, cuándo y con quién. A través de estas aplicaciones un usuario deja mucho en su dispositivo, sin muchas veces darse cuenta de esto. Dejamos las fotos y videos con nuestra familia y amigos, los datos personales de todos nuestros contactos, todas nuestras cuentas de email personales y laborales, todos los eventos a los que planeamos ir, turnos con el médico, direcciones, notas, etc.

Además de todos estos datos personales que nosotros de alguna forma acordamos dejar en nuestro dispositivo, mucha gente no es consciente que un dispositivo que se conecta a una red WiFi puede detectar, entre otras cosas, dónde se encuentra la persona, además de tener un completo historial de dónde estuvo y qué recorrido hizo hasta llegar al punto donde se encuentra, gracias a la lista de redes donde estuvo conectado el dispositivo. También

hay aplicaciones que tienen acceso a datos del dispositivo que pueden identificar unívocamente al dispositivo y a su dueño.

La lista de datos personales nuestros y de nuestros conocidos que dejamos que nuestro dispositivo manipule es muy larga y sigue aumentando. Si juntamos toda la información que guardamos y aquella información que muchas veces no notamos está en el dispositivo, le estamos dando demasiado poder a quien tenga acceso a la misma.

1.2.4. Trabajos relacionados

Distintos enfoques han sido implementados con el fin de restringir abusos por parte de distintas aplicaciones. Algunas niegan completamente el acceso al recurso solicitado [**12**] [**6**], poniendo en riesgo el correcto funcionamiento de la aplicación solicitante y la experiencia del usuario. Por ejemplo, podríamos denegar el acceso a internet a un videojuego en tiempo de ejecución. El problema con este enfoque, es que las aplicaciones suelen fallar cuando se les niega el acceso a un recurso solicitado. Esto puede provocar un funcionamiento incorrecto de la misma, afectar la experiencia del usuario y hasta puede provocar que la aplicación se cierre abruptamente.

Otras implementaciones utilizan datos falsos polimórficos a los que la aplicación espera, reemplazando los recursos solicitados en lugar de denegar el acceso a los mismos [**3**] [**2**]. Un ejemplo típico es devolver una agenda vacía de contactos en vez de la lista de contactos real del usuario o una foto en blanco cuando una aplicación con acceso a la cámara nos pida una imagen. De esta forma, cierta información que el usuario considera privada no podrá ser accedida por una aplicación que no es confiable y permanecerá protegida. Otro ejemplo es modificar la ubicación del celular para que muestre una distinta a la real. Esto evita que la aplicación falle pero muchas veces se vuelve inútil porque su funcionalidad perdió sentido. Por ejemplo, ¿para qué queremos usar *WhatsApp* si le vamos a dar una lista vacía de contactos?

Algunas de estas estrategias se basan en la reconfiguración del código fuente de la aplicación para lograr un control más fino de la misma en tiempo de ejecución [**6**] [**4**] [**7**]. También hay versiones modificadas del sistema operativo de Android [**2**] [**3**] que permiten este mismo tipo de controles, pero muchas veces instalar una nueva versión en nuestro dispositivo no es una opción por limitaciones de hardware, software o desconocimiento del usuario.

Finalmente, en el trabajo expuesto en un artículo de nombre *TISSA* [**5**] se propone permitir al usuario configurar para cada aplicación y cada recurso sobre la cual ésta tenga permisos, la opción de reemplazar dichos recursos con datos polimórficos vacíos, falsos, anónimos o auténticos. Es un enfoque más flexible pero sigue teniendo el mismo problema mencionado previamente, donde el control sobre algunas aplicaciones hace que la misma pierda sentido. Sin embargo, este último enfoque parece ser el más cercano

al que nosotros proponemos.

Las ideas previamente expuestas son buenas para resolver el problema de asegurar que los datos que el usuario considera privados no sean accedidos por las aplicaciones en cuestión, pero son propensas a fallo y provocan que una aplicación deje de funcionar según lo espera el usuario que, a pesar de estar preocupado por la seguridad de sus datos, quiere usar la aplicación y que la misma funcione correctamente.

1.3. Objetivos del trabajo

El objetivo de este trabajo es construir un sistema que, basado en los pros y contras de las soluciones ya propuestas, mejore la experiencia del usuario sin dejar de proteger su privacidad. Nuestra solución es independiente del sistema de permisos de Android. La idea es definir para cada recurso sensible que exponga información privada del usuario, una granularidad que nos permita balancear la usabilidad y el acceso de una aplicación a nuestra información.

Para esto definimos un control de recursos que en lugar de devolver datos falsos o anónimos, devuelva un subconjunto predefinido por el usuario de los datos que la aplicación requiere. Por ejemplo, nosotros definimos la granularidad de los contactos en base al sistema de grupos de Android. Gracias a eso un usuario podría decidir que *WhatsApp* tenga acceso sólo a mi grupo de familia y amigos y no al laboral.

Este nuevo enfoque tiene algunas ventajas con respecto a los otros:

Nuestra solución permite que un usuario pueda tomar distintas decisiones según una aplicación dada y un recurso determinado. Por ejemplo, si yo uso la aplicación de *LinkedIn* sólo por motivos laborales, podría limitar su acceso a mi agenda de contactos para simular que sólo están mis compañeros de trabajo en esa agenda y no la totalidad de mi agenda.

Otra ventaja, es que una aplicación no deja de funcionar al limitar el acceso a los recursos con un subconjunto de los datos reales por lo que la experiencia del usuario mejora significativamente respecto del resto de los enfoques antes planteados. Por un lado, no tenemos una aplicación que se rompe por denegarle el acceso al recurso en tiempo de ejecución cuando ésta no se lo esperaba, ni una aplicación como *WhatsApp* con una agenda vacía, lo cual parece no tener mucho sentido.

1.4. Estructura del documento

En lo que queda de este documento el lector encontrará una sección con las bases teóricas necesarias para poder entender a fondo el trabajo presentado, la prueba de concepto realizada a comienzos del mismo junto con sus conclusiones, una sección dedicada a la arquitectura del sistema propuesto,

experimentaciones que se realizaron una vez que el sistema fue implementado y una sección de conclusiones finales, además de trabajos que planteamos para el futuro y la bibliografía pertinente.

1.5. Términos y abreviaciones

1.5.1. Hook o hooking

En español gancho, en programación, el término *hooking* cubre un rango de técnicas utilizadas para alterar o aumentar el comportamiento de un sistema operativo, aplicaciones, o de otros componentes de software interceptando llamadas a funciones, mensajes o eventos que permiten que dichos componentes interactúen. El código que maneja estas funciones, eventos o mensajes interceptados es denominado *hook*.

El mismo tiene muchos usos, incluyendo depuración de programas o extensión de alguna funcionalidad, además de ser usado por código malicioso.

1.5.2. Query o consulta

En este informe utilizaremos de forma intercambiable los dos términos refiriéndonos a un pedido preciso para recuperar datos de una base de datos.

1.5.3. URI: Identificador Uniforme de Recursos

Uniform Resource Identifier (identificador uniforme de recursos, en español), identifica un recurso abstracto o físico según se encuentra especificado en RFC 2396 [**17**]. En el caso particular de este trabajo, una URI identifica datos dentro de un proveedor de contenido, como ser el que maneja los datos de contactos en un dispositivo Android.

1.5.4. Java Reflection

En informática, reflexión (o reflexión computacional) es la capacidad que tiene un programa para observar y opcionalmente modificar su estructura de alto nivel. Entre otras cosas, nos permite examinar la clase de un objeto, construir un objeto para una clase, examinar un campo o método de una clase o invocar cualquier método de un objeto, todo en tiempo de ejecución.

Java Reflection es comúnmente usado por programas que requieren la capacidad de examinar y modificar el comportamiento de ciertas aplicaciones corriendo en la máquina virtual de Java, en tiempo de ejecución. Es una técnica poderosa que permite que aplicaciones que la usan realicen operaciones que de otra forma les sería imposible realizar.

1.5.5. SQLite

Es un sistema de gestión de bases de datos relacional compatible con ACID (Atomicity, Consistency, Isolation and Durability: Atomicidad, Consistencia, Aislamiento y Durabilidad) y usada por el sistema operativo Android para almacenar muchos de los recursos del equipo. Está contenida en una relativamente pequeña (∼275 kiB) biblioteca escrita en C.

1.5.6. Stack trace

Llamamos stack trace (pila de ejecución, en español) a un reporte de los frames activos en la pila en un cierto punto en el tiempo durante la ejecución de un programa.

1.5.7. Activity

Activity es una de las clases principales de una aplicación con interfaz gráfica en Android. Es un componente de una aplicación que provee una pantalla con la cual el usuario puede interactuar para realizar una acción tal como discar un número telefónico, sacar una foto, enviar un email, o ver un mapa. Cada actividad tiene una ventana en la cual dibujar su interfaz de usuario. La ventana usualmente llena la pantalla, aunque podría ser más pequeña o *flotar* en la parte superior de otra ventana.

1.5.8. Time-of-check Time-of-use (TOCTOU)

En desarrollo de software, llamamos time of check to time of use (TOCTOU) del inglés, momento de verificación, momento de uso a una clase de bug de software causado por cambios en el sistema entre que se verifica una condición (como ser una credencial de seguridad) y el uso del resultado de esa verificación. Este es un ejemplo de race condition (condición de carrera, en español). La Condición de carrera se da principalmente cuando varios procesos acceden al mismo tiempo a un recurso compartido, por ejemplo una variable, cambiando su estado y obteniendo de esta forma un valor no esperado de la misma.

1.5.9. ADB, Android Debug Bridge

Es una herramienta versátil de línea de comando que nos permite comunicarnos con una instancia de un dispositivo emulado o un dispositivo Android conectado a nuestra PC. Es un programa cliente-servidor que incluye tres componentes:

- Un cliente que corre en nuestra computadora de desarrollo. Este cliente se puede invocar desde una consola a través de un comando adb.

- Un servidor que corre como proceso en segundo plano en nuestra computadora de desarrollo. Este servidor maneja la comunicación entre el cliente y el demonio adb que corre en el emulador o el dispositivo.
- Un demonio (daemon, en inglés) que corre como proceso en segundo plano en cada instancia de emulador o dispositivo.

Cuando iniciamos el cliente adb, éste primero se cerciora que haya un servidor corriendo. Si no hay, levanta el proceso servidor. El servidor corre en el puerto TCP 5037 y escucha los comandos enviados desde dos puertos a los que se conecta el cliente para comunicarse con el servidor adb. Luego, el servidor prepara las conexiones para cada emulador o dispositivo corriendo, escaneando los puertos a los que estos podrían estar conectados.

Es importante notar que, gracias a adb, tenemos un acceso directo a nuestro emulador o dispositivo, por lo cual tenemos acceso a su sistema operativo, filesystem y a los comandos del mismo, como por ejemplo, la posibilidad de cambiar los permisos de distintos archivos como ser las bases de datos del sistema o de alguna aplicación en particular, simplemente utilizando el comando *chmod*.

1.5.10. LogCat

Es el sistema de logging de Android que provee un mecanismo para colectar y ver la salida del sistema de debug. Logs de varias aplicaciones e incluso porciones del sistema son colectados en una serie de buffers circulares, que luego pueden ser inspeccionados y filtrados por comandos *LogCat*. *LogCat* descarga un log de los mensajes del sistema, que incluye cosas como los stack traces cuando se lanza una excepción y los mensajes que escribimos en nuestra aplicación usando la clase *Log*. Además, podemos usar *LogCat* desde la consola de *ADB* para ver los mensajes logeados en tiempo real. De esta forma podemos organizar nuestros logs utilizando palabras claves y tipo de mensaje: error, advertencia, información, etc.

1.5.11. APK

Un archivo con extensión .apk (Application Package File) es un paquete para el sistema operativo Android. Este formato es una variante del formato JAR de Java y se usa para distribuir e instalar componentes empaquetados para la plataforma Android para smartphones y tablets.

1.5.12. JAR

Un archivo JAR (por sus siglas en inglés, Java ARchive) es un tipo de archivo que permite empaquetar aplicaciones escritas en el lenguaje Java. Las siglas están deliberadamente escogidas para que coincidan con la palabra inglesa "jar"(tarro). Los archivos JAR están comprimidos con el formato ZIP y cambiada su extensión a .jar. Existen tres operaciones básicas con

este tipo de archivos: ver contenido, comprimir y descomprimir.

1.5.13. GSM y CDMA

GSM y CDMA son dos estándares de comunicación inalámbrica que actualmente dominan el mercado. GSM es el acrónimo de Global System for Mobile Communications o sistema global para comunicaciones móviles, y el CDMA, Code Division Multiple Access o división de código con acceso múltiple.
Ambos estándares actúan cómo un juego de reglas para dispositivos inalámbricos; estas reglas dictan como se envía la información y sobre qué frecuencia de radio. Los celulares GSM se identifican por llevar en su interior tarjetas SIM (módulo de identidad del suscriptor) y los celulares CDMA solamente obtienen comunicación por una configuración directa.

1.5.14. UMTS

Sistema universal de telecomunicaciones móviles (Universal Mobile Telecommunications System) es una de las tecnologías usadas por los móviles de tercera generación, sucesora de GSM, debido a que la tecnología GSM propiamente dicha no podía seguir un camino evolutivo para llegar a brindar servicios considerados de tercera generación como ser: capacidades multimedia, una velocidad de acceso a Internet elevada, la cual también le permite transmitir audio y video en tiempo real; y una transmisión de voz con calidad equiparable a la de las redes fijas.

Capítulo 2

Base teórica

2.1. Conceptos previos [10]

2.1.1. Esquema de seguridad de Android

Android provee una plataforma de código abierto y un entorno para aplicaciones en dispositivos móviles. Las aplicaciones Android hacen uso de hardware y software avanzado, así como datos de usuario. Para proteger todos estos recursos, Android debe ofrecer un ambiente para las aplicaciones que afirme la seguridad del usuario, datos, aplicaciones, el dispositivo y la red.

Android, basado en Linux, fue diseñado con un sistema de seguridad multi-capa que provee la flexibilidad requerida para una plataforma abierta, proveyendo al mismo tiempo protección a todos sus usuarios.

Los principales pilares sobre los cuales está construido Android son:

- Hardware del dispositivo: Android corre sobre distintas configuraciones incluyendo teléfonos inteligentes y tablets, pero no hace uso de ninguna capacidad de seguridad específica del hardware.
- Sistema Operativo Android: el núcleo del sistema operativo está construido sobre el kernel de Linux. Todos los recursos del dispositivo, como ser las funciones de la cámara, los datos del GPS, las funciones de Bluetooth, del teléfono, las conexiones de red, etc. son accedidas a través del sistema operativo.
- Entorno de ejecución: las aplicaciones Android son en su mayoría escritas en código Java y corren en la máquina virtual Dalvik. Sin embargo, muchas aplicaciones, incluyendo los servicios y aplicaciones del núcleo de Android son aplicaciones nativas o incluyen bibliotecas nativas. Tanto Dalvik como las aplicaciones nativas corren dentro del mismo ambiente de seguridad, contenido en un Sandbox. Cada aplicación tiene una parte dedicada del sistema de archivos en el cual puede escribir sus datos privados, incluyendo bases de datos y archivos.

Las aplicaciones de Android extienden el núcleo del sistema operativo. Hay dos fuentes principales de aplicaciones:

- Aplicaciones pre-instaladas: Android incluye un conjunto de aplicaciones pre-instaladas como ser teléfono, email, calendario, navegador web, y agenda de contactos. Dichas aplicaciones funcionan tanto como aplicaciones de usuario como para proveer capacidades puntuales

del dispositivo que pueden ser accedidas por otras aplicaciones. Estas aplicaciones pueden ser parte de la plataforma de código abierto de Android, aplicaciones desarrolladas por los fabricantes de un dispositivo particular o incluso las operadoras telefónicas.
- Aplicaciones instaladas por el usuario: Android provee un ambiente de desarrollo abierto que soporta cualquier aplicación de terceros. En *Google Play* y otros sitios se ofrece a los usuarios cientos de miles de aplicaciones.

2.1.2. Arquitectura de seguridad en Android

Android propone controles de seguridad tradicionales para su sistema operativo para:

- Proteger los datos de usuario.
- Proteger los recursos del sistema (incluyendo la red).
- Provee aislamiento de aplicaciones.

Para lograr estos objetivos, provee las siguientes características claves de seguridad:

- Seguridad robusta a nivel de sistema operativo a través del kernel de Linux.
- Sandbox obligatorio para todas las aplicaciones.
- Firma de aplicaciones.
- Sistema de permisos definidos por las aplicaciones y concedidos por los usuarios.

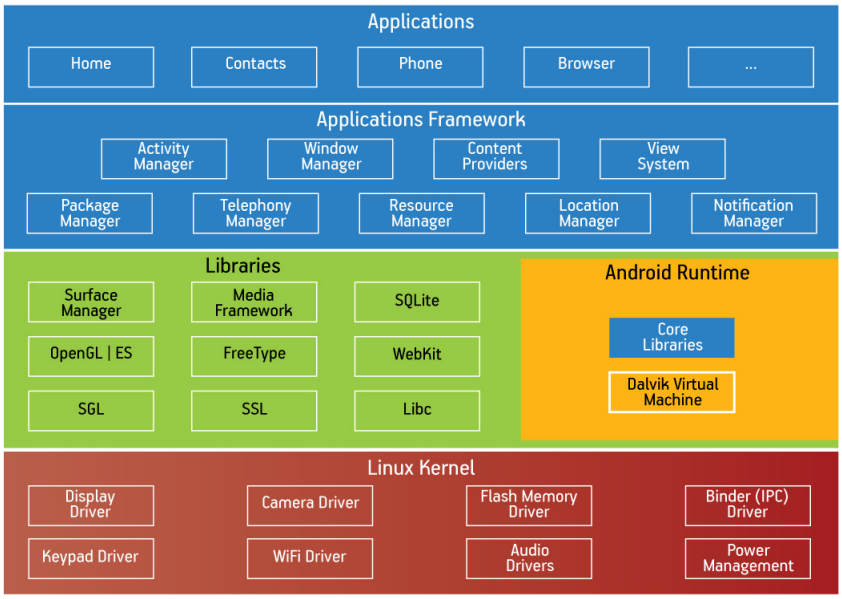


FIGURA 2. Capas de Android

En la figura 2 se resumen los componentes de seguridad y los distintos niveles del stack de Android. Cada componente asume que los componentes por debajo son apropiadamente seguros. Con excepción de una pequeña

cantidad de código del sistema operativo Android que corre como root, todo el código sobre el kernel de Linux es restringido por el sandbox de su aplicación.

2.1.3. Seguridad en Linux

Android tiene sus cimientos sobre el kernel de Linux. El mismo ha sido usado por años, y es usado por millones de entornos donde la seguridad es importante. A través de su historia de constante investigación, ataques y reparaciones por miles de desarrolladores, Linux se convirtió en un kernel estable y seguro usado por muchas corporaciones y profesionales de seguridad.

El kernel de Linux provee a Android varias características de seguridad claves, incluyendo:

- Un modelo de permisos basados en usuarios.
- Aislamiento de procesos.
- Mecanismo extensible para IPC seguro.
- La capacidad de remover partes del kernel innecesarias y potencialmente inseguras.

Por ser un sistema operativo multiusuario, un objetivo de seguridad fundamental del kernel de Linux es el de aislar uno de otro los recursos de usuario. La filosofía de seguridad de Linux es proteger los recursos de usuario uno de otro. Por consiguiente, Linux:

- Impide que un usuario A lea los archivos de un usuario B.
- Asegura que un usuario A no agote la memoria de un usuario B.
- Asegura que un usuario A no agote los recursos de CPU de un usuario B.
- Asegura que un usuario A no agote los dispositivos de un usuario B (por ejemplo GPS o bluetooth).

2.1.4. Sandbox: aislamiento de procesos

El concepto *sandboxing* se refiere a que las aplicaciones instaladas en un dispositivo Android son cuidadosamente controladas respecto a lo que pueden hacer al sistema y a otras aplicaciones: cada aplicación corre en su propio *sandbox* (arenero, en español), con un impacto mínimo en los areneros del resto de las aplicaciones.

Android aprovecha de la protección basada en usuario de linux identificando y aislando los recursos de las aplicaciones. El sistema de Android asigna un único ID de usuario (UID) a cada aplicación y la corre como si fuese un proceso separado, salvo que una aplicación pida explícitamente correr en el mismo espacio de otra con la que comparta la firma. Esto difiere de otros sistemas operativos (incluso de la configuración tradicional de Linux), donde múltiples aplicaciones corren con los mismos permisos de usuario. De esta forma el kernel implementa un Sandbox de aplicación.

Por defecto, las aplicaciones no pueden interactuar entre ellas y tienen acceso limitado al sistema operativo. Si la aplicación A trata de hacer algo malicioso como leer los datos de la aplicación B o realizar una llamada telefónica sin permisos (lo cual se hace a través de una aplicación separada), entonces el sistema operativo protege contra esto, dado que la aplicación A no tiene los privilegios de usuario apropiados. El uso de sandbox es simple, auditable, y basado en décadas de uso de separación de procesos y permisos sobre archivos.

Dado que el Sandbox de aplicación está en el kernel, este modelo de seguridad se extiende a código nativo y aplicaciones del sistema operativo. Todo el software por encima del kernel en la figura 2, incluyendo las bibliotecas del sistema operativo, el framework de aplicaciones, las aplicaciones en tiempo de ejecución, y todas las aplicaciones corren dentro del Sandbox.

Como toda característica de seguridad, el Sandbox de aplicación no es irrompible. Sin embargo, al quebrarlo debemos tener en cuenta que se puede comprometer la seguridad del kernel de Linux.

Existen mecanismos para que una aplicación pueda salirse de su sandbox para comunicarse con otras aplicaciones o el sistema operativo, mediante el uso de permisos o a través de IPC.

2.1.5. Directorio *system*

El directorio *system* contiene el kernel de Android así como las bibliotecas del sistema operativo, las aplicaciones y demás archivos que las mismas necesitan para su ejecución, entre otros. Esta partición es de sólo lectura. Cuando un usuario inicia su dispositivo en *Modo Seguro*, sólo las aplicaciones del núcleo de Android están disponibles. Esto asegura que un usuario pueda iniciar su teléfono en un ambiente libre de aplicaciones de terceros.

2.1.6. Permisos del sistema de archivos

En un ambiente UNIX, los permisos del sistema de archivos aseguran que un usuario no puede alterar o leer los archivos de otro usuario.

En el caso de Android, cada aplicación corre con su propio usuario. A menos que el desarrollador explícitamente exponga archivos a otras aplicaciones, los archivos creados por una aplicación no pueden ser leídos o alterados por otras aplicaciones.

En el caso de la tarjeta SD no tenemos sistema de control de acceso dado que se usa FAT como sistema de archivos. Esta decisión de diseño de Android puede llevar a graves problemas de seguridad al permitir a una aplicación leer y/o escribir cualquiera de los archivos almacenados de la SD. Por ejemplo una aplicación podría comprometer la integridad de archivos de alguna otra.

2.1.7. Permisos de root en Android

Por defecto, en Android sólo el kernel y un pequeño subconjunto de aplicaciones del núcleo corren con permiso de root. Android no impide que un usuario o aplicación con permisos de root pueda modificar el sistema operativo, el kernel o alguna otra aplicación.

En general, root tiene acceso completo a todas las aplicaciones y a todos los datos. Los usuarios que cambian los permisos en un dispositivo Android para conceder acceso a root a aplicaciones aumentan la exposición a aplicaciones maliciosas y potenciales fallos en las aplicaciones.

En el contexto del sistema operativo Android, llamamos *rooting* al proceso de permitir a usuarios de smartphones, tablets y otros dispositivos alcanzar control privilegiado, conocido como *acceso* root dentro del subsistema de Android.

El *Rooting* es usualmente realizado con la meta de vencer las limitaciones que las operadoras telefónicas o los fabricantes de hardware pusieron a algunos dispositivos, resultando en la capacidad de alterar o reemplazar aplicaciones del sistema y configuraciones, correr aplicaciones especializadas que requieren permisos de nivel de administrador, o realizar otras operaciones que de otra forma serían inaccesibles a un usuario tradicional. En Android, el *rooting* puede también facilitar la eliminación completa o reemplazo del sistema operativo del dispositivo, usualmente con una edición más reciente del sistema operativo actual.

Como Android proviene del kernel de Linux, rootear un dispositivo es similar a obtener permisos administrativos en Linux o cualquier otro sistema operativo basado en Unix.

2.1.8. Seguridad a nivel aplicación: modelo de permisos

Todas las aplicaciones de Android corren en un Sandbox, como fue descripto anteriormente en este documento. Por defecto, una aplicación Android sólo puede acceder a un rango limitado de recursos del sistema. El sistema maneja el acceso a recursos que, de ser usados de forma incorrecta o maliciosa, podrían impactar negativamente en la experiencia del usuario, la red o los datos del dispositivo.

Estas restricciones son implementadas de distintas formas. Algunos atributos del sistema son restringidos con la falta intencional de funciones de API para acceder a funcionalidades sensibles (por ejemplo, no existe una API de Android para manipular directamente la tarjeta SIM). En algunos casos, la separación de roles provee una medida de seguridad, como con la separación del almacenamiento por aplicación. En otros casos, las APIs sensibles son sólo accedidas por aplicaciones de confianza y protegidas a través

de un mecanismo de seguridad conocido como *Permisos*, algunos de los cuales son mapeados a grupos de Linux.

Entre dichas APIs protegidas se encuentran:

- Funciones de la cámara.
- Datos de ubicación (GPS).
- Funciones de bluetooth.
- Funciones del teléfono.
- Funciones de SMS/MMS.
- Conexiones a la red y a los datos.

Estos recursos son sólo accesibles a través del sistema operativo. Para hacer uso de las APIs protegidas en el dispositivo, una aplicación debe definir las funcionalidades que necesita en su manifiesto. Cuando la aplicación está por instalarse, el sistema muestra un diálogo al usuario que indica los permisos requeridos y pregunta si se quiere proseguir con la instalación. Si el usuario continúa con la instalación, el sistema acepta que el usuario ha concedido todos los permisos requeridos. El usuario no puede conceder o denegar permisos individuales; debe conceder o denegar todos los permisos solicitados como un bloque.

Una vez concedidos, los permisos son aplicados a la aplicación mientras esté instalada. Para evitar confusión por parte del usuario, el sistema no notifica al usuario otra vez el permiso concedido a la aplicación. Una aplicación que está incluida en el núcleo del sistema operativo o que fue incluida por el fabricante no solicita permisos al usuario. Los permisos son retirados una vez que la aplicación es desinstalada, por lo que en subsecuentes re-instalaciones, se volverán a mostrar los permisos requeridos al usuario.

Dentro de la configuración del dispositivo, el usuario puede ver los permisos por aplicación instalada. Los usuarios pueden también apagar algunas funcionalidades globalmente cuando eligen, por ejemplo, deshabilitar el GPS, la radio o el WiFi.

En caso de que una aplicación intente acceder una funcionalidad protegida que no ha sido declarada en su manifiesto, resultará en una falla de permisos que típicamente lanzará una excepción de seguridad de vuelta a la aplicación. Un ejemplo de visualización de permisos por aplicación se muestra en la figura 1.

Existen permisos por defecto del sistema y las aplicaciones pueden declarar sus propios permisos para que otras aplicaciones usen. También existen algunas funcionalidades que no están disponibles para aplicaciones de terceros, pero pueden ser usadas por aplicaciones pre-instaladas por el fabricante.

Ya nombramos los principales problemas del sistema de permisos: son estáticos, es decir, una vez instalada la aplicación, no podemos revocarle algún permiso, además que un usuario debe conceder todos los permisos declarados por la aplicación para que pueda ser instalada. Muchas veces

las aplicaciones no aplican el principio de menor privilegio, que promueve a que acceda solo a la información y recursos necesarios para su legitimo propósito. Sumado a que una actualización puede pedir permisos adicionales a los permisos pedidos originalmente.

2.1.9. Comunicación inter-proceso (IPC)

Los procesos pueden comunicarse usando cualquiera de los mecanismos tradicionales de UNIX. Algunos ejemplos son el sistema de archivos, sockets locales, o signals (señales, en español).

Android también provee nuevos mecanismos IPC:

- Binder: mecanismo para llamada a un proceso remoto diseñado para una comunicación de alto desempeño cuando se realizan llamadas dentro o a través de procesos.
- Servicios: (discutido más arriba) puede proveer una interfaz de acceso directo utilizando un binder.
- Intents: un intent es un simple mensaje que representa la *intención* de hacer algo. Un intent puede ser usado también para transmitir eventos interesantes, como una notificación, a través del sistema.
- ContentProviders (proveedor de contenido, en español): es un almacén de datos que provee acceso a datos en el dispositivo; el ejemplo clásico es el del ContentProvider que es usado para acceder la lista de contactos del usuario. Una aplicación puede acceder datos que otras aplicaciones han expuesto vía un ContentProvider, y una aplicación puede también definir su propio ContentProvider para exponer datos propios.

Se fomenta que el desarrollador use las mejores prácticas al momento de manejar los datos del usuario y que evite la inserción de vulnerabilidades de seguridad.

El mecanismo de IPC no tiene control sobre la información que es compartida entre las distintas aplicaciones, posibilitando que una aplicación obtenga información privada sin contar con el permiso para accederla. Por ejemplo, si tengo dos aplicaciones, una que lee mi agenda de contactos y otra que se conecta a internet, juntas podrían subir mis contactos a algún servidor externo sin que este proceso sea detectado por el usuario.

2.2. Cydia Substrate [9]

Cydia Substrate es una API creada por Jay Freeman, conocido como *Saurik* que salió en mayo del 2013 para el sistema operativo Android y está actualmente disponible en *Google Play* de forma gratuita.

Es una herramienta principalmente para desarrollo, que nos permite fácilmente modificar código en aplicaciones de las cuales no tenemos el código fuente y no fueron creadas por nosotros. De esta forma podemos modificar

esas aplicaciones e incluso personalizar el sistema por completo.

La desventaja que tiene el uso de *Cydia Substrate* es que, por un lado, necesitamos un teléfono rooteado, a pesar de que esto es de esperarse, y por otro, la extensión de código corre con los mismos permisos con los que corre la aplicación que originalmente hizo la llamada que estamos modificando.

Por otro lado, *Cydia Substrate* nos permite extender código ya existente mediante simples hooks de clases y métodos de nuestro interés, sin necesidad de hacer ingeniería reversa. Para poder lograr esto, nos provee de algunos servicios básicos entre los cuales encontramos:

2.2.1. MS.hookClassLoad

A diferencia de muchos lenguajes de programación, como *Objective-C*, donde todo el código es cargado y todas las clases están disponibles en el instante en que la imagen es cargada, Java gradualmente carga código en varios *class loaders* (cargadores de clase). Esto significa que para poder, por ejemplo, hookear un método de una clase, primero necesitamos saber si la clase está disponible.

Es por eso que *Cydia Substrate* provee una API que nos permite ser notificados al momento en que una clase dada es cargada, dándonos un puntero al entorno en el cual la clase fue cargada así como una referencia a dicha clase.

Esta API tiene un objeto que implementa una interfaz simple *MS.ClassLoadHook* que tiene un sólo método llamado *classLoaded* el cual es ejecutado cuando la clase es cargada. Este método espera como argumento la clase Java que fue cargada.

2.2.2. MS.hookMethod

La tarea más usual del desarrollador que usa *Cydia Substrate* para realizar una extensión al sistema es modificar el funcionamiento de código existente. Esto implica poder reemplazar cualquier método así como llamar a su implementación original. *Cydia Substrate* permite al desarrollador proveer un callback que será ejecutado en lugar del método original. Este callback es un objeto que implementa una interfaz *MS.MethodHook*, que tiene un único método llamado *invoked* que típicamente usa una clase anónima. Para ejecutar la implementación original, o si la modificación es simplemente alterar los argumentos que recibía la misma, una instancia de *MS.MethodPointer* es recibida durante el hook; este objeto tiene un método llamado *invoke* que puede ser usado para llamar al código original.

El desarrollador puede elegir entre pasar un solo argumento del tipo *MS.MethodAlteration*. Esta clase implementa *MS.MethodHook* y extiende

MS.MethodPointer, permitiendo al desarrollador evitar tener que declararlos de forma separada.

En el caso de que múltiples desarrolladores hookeen la misma API, las llamadas son apiladas: el último hook tendrá una *implementación original* que sería el hook anterior, que a su vez tendrá una *implementación original* de acuerdo a quién sea que hookeó antes, hasta eventualmente encontrar el método original.

2.3. Recursos

2.3.1. Introducción

Como mencionamos anteriormente, el sistema de administración de permisos de Android requiere que los permisos de las aplicaciones sean especificados en su manifiesto. Cuando la aplicación es instalada, Android notifica al usuario de los permisos y recursos que la aplicación necesita. Una vez instalada, sólo se controla que se accedan a los recursos especificados en su manifiesto.

Los trabajos relacionados presentados en el capítulo anterior muestran una forma de poder tener un control más fino de los recursos. Muchos de ellos plantean la posibilidad de denegar el acceso de una aplicación a un recurso mientras que otros crean recursos falsos con los cuales las aplicaciones puedan interactuar sin fallar.

La base de nuestro sistema de control de recursos parte de no crear recursos falsos polimórficos a los originales, sino permitir el acceso a los recursos a un nivel más fino que el de todo o nada. Para lograr esto, es necesario poder definir para cada recurso qué significa tener un nivel de granularidad más fino sobre el recurso. Además, dicho control debe ser transparente a la aplicación que accede a los recursos.

Naturalmente, asumimos que las aplicaciones de terceros no pueden acceder a los recursos del sistema (por ejemplo, el sistema de archivos) o a los datos privados de otra aplicación de forma directa; sólo lo pueden hacer a través de la API provista por Android. Sin embargo, somos conscientes de que cualquier aplicación con permisos de root puede no sólo pasar por sobre la API de Android sino además hacer y deshacer a su antojo, pero decidimos no considerar estos casos de momento y dejarlos para trabajos posteriores.

Las aplicaciones del sistema y las instaladas por el fabricante, aquellas que vienen preinstaladas apenas adquirimos el dispositivo, también son administradas por el sistema propuesto. En algunos trabajos relacionados, como es el caso de TISSA [**5**], han decidido que dichas aplicaciones queden fuera del alcance de su control de recursos.

Durante el desarrollo del sistema que, de ahora en más denominaremos ASA (del acrónimo Android Security Agency que en inglés significa Agencia

de seguridad de Android), nos encontramos con varios recursos de interés para los cuales controlar el acceso parecía ser relevante. Entre ellos están:

- Agenda de contactos.
- Datos de teléfono que identifican al dispositivo y a su dueño.
- Log de llamadas.
- Archivos que se encuentran en la memoria externa del dispositivo.
- Calendarios.
- Información proveniente de redes inalámbricas.
- Información proveniente de bluetooth.
- Historial de navegación.
- Casilla de SMSs.
- Información de la ubicación del equipo.
- Acceso a la cámara.
- Información de las aplicaciones instaladas en el dispositivo.

Entre todos ellos elegimos un subconjunto que nos pareció lo suficientemente bueno y variado para empezar. El primero es el acceso a los datos de la agenda de contactos del dispositivo, luego modificamos el acceso a datos del dispositivo que identifican al mismo y a su dueño, también modificamos el acceso a datos relacionados con la configuración de las redes inalámbricas y, finalmente, modificamos todas las formas de determinar la ubicación del dispositivo.

Elegimos estos 4 recursos debido a que el dominio en el que se maneja cada uno es muy distinto. En el caso de la agenda de contactos, éste es un recurso común entre todos los dispositivos móviles y es esperable que la información proveniente de la agenda de contactos de un dispositivo Android sea información que queremos resguardar de aplicaciones maliciosas pues compromete datos privados de otras personas, además de los de uno mismo. Por otro lado, la información proveniente de los datos del teléfono es utilizada por muchas aplicaciones para identificar unívocamente a un usuario por razones de mercado o maliciosas. La información proveniente del recurso de redes inalámbricas tiene dos puntos que pueden comprometer la privacidad de un usuario: por un lado, un dispositivo puede ser identificado utilizando, por ejemplo, su dirección MAC y, por otro lado, información de las redes inalámbricas puede ser utilizada para localizar al equipo mediante las redes escaneadas. Finalmente, el recurso de geolocalización es de suma importancia para cualquier persona que quiera controlar qué aplicación tiene acceso a su ubicación.

De forma general, los proveedores de contenido y servicios que elegimos controlar con ASA son:

- ContactsProvider: proveedor de contenido de contactos.
- TelephonyManager: provee acceso a información de servicios de telefonía del dispositivo, así como su estado y demás información relacionada.
- WifiManager: provee funcionalidad principal para manejar todos los aspectos de la conectividad WiFi del dispositivo.

- LocationManager: provee acceso a servicios del sistema de ubicación. Estos servicios permiten obtener actualizaciones periódicas de la ubicación geográfica del dispositivo o recibir un aviso cuando el mismo entra en las proximidades de una ubicación dada.

Por completitud, se modificaron otros servicios, incluyendo algunos de la API interna de Android que permitían acceso a los recursos considerados en nuestro sistema. Se dará detalle de los mismos en las secciones dedicadas a cada recurso particular.

2.3.2. Agenda de contactos

El proveedor de contactos es el repositorio principal de la información de contactos del usuario, incluyendo datos de aplicaciones de contactos o aplicaciones sociales como ser *Facebook*, *Twitter*, etc.

Para acceder a este recurso, una aplicación debe tener en su manifiesto declarado el permiso *READ_CONTACTS* o bien *WRITE_CONTACTS*. Una vez instalada y gracias a este permiso, una aplicación puede consultar la lista de contactos del dispositivo, obtener los detalles para un contacto en particular, o modificar los datos de un contacto.

Un ejemplo típico de cómo una aplicación con permiso para acceder a contactos realiza una query al recurso:

```
Cursor groups = context.getContentResolver().query(
    ContactsContract.Groups.CONTENT_URI,
                                    new String[] { ContactsContract.
                                        Groups._ID }
                                    , null, null, null);
```

En este ejemplo, dado un contexto, pedimos el *ContentResolver* contra el que realizamos una query a la base de datos de grupos para pedir todos los identificadores de grupo del sistema. El *ContentResolver* resuelve a partir de la URI, identificador que recibe como primer parámetro en la llamada a query, quién es el proveedor de contenido que puede resolver la consulta (en este caso se trata de *ContactsProvider*).

2.3.2.1 ContentProvider y ContentResolver

Por defecto, una aplicación Android no puede acceder a los datos de otra aplicación. Éste es uno de los pilares fundamentales de la seguridad del sistema operativo, el principio de *sandboxing*. Frecuentemente, queremos que una aplicación comparta datos con otras. Los *Content Providers* (proveedor de contenido, en español) manejan el acceso centralizado a un repositorio de datos.

Algunos ejemplos de *Content Providers* del sistema operativo Android son:

- Contacts: maneja los datos de contactos.
- MediaStore: maneja los datos de media, como ser música, fotos, videos, etcétera.

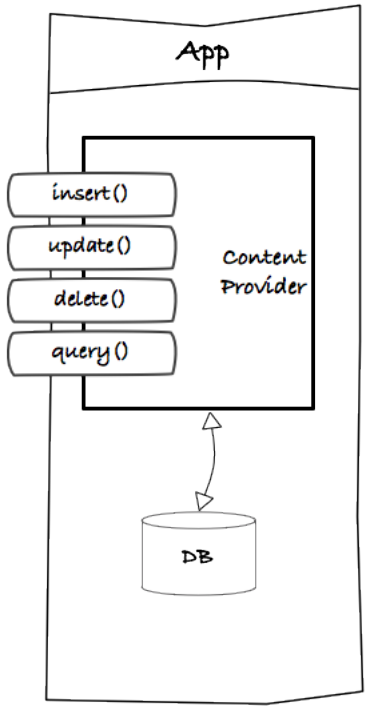


FIGURA 3. Resumen de ContentProvider

- Bookmarks: interfaz a los datos de marcadores.
- Settings: contiene las preferencias globales a nivel de sistema.

Un ejemplo típico de cómo el sistema de contactos puede trabajar en Android comprende dos aplicaciones: una responsable de la interfaz de usuario y la otra responsable de los datos 4.

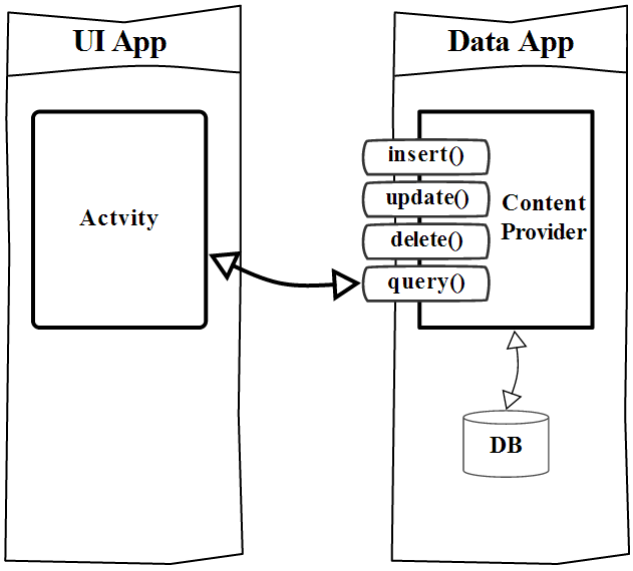


FIGURA 4. Uso típico de ContentProvider

Un *ContentProvider* es iniciado la primera vez que es usado, a través de una llamada a *onCreate()*. Un *ContentProvider* puede recibir pedidos originados desde otros procesos. En este caso, los métodos que responden a esos pedidos, por ejemplo, *query*, *insert*, *delete* o *update*, son llamados desde un pool de threads que corren en el proceso de *ContentProvider*. Por otra parte, en caso de que el pedido venga desde el mismo proceso del *ContentProvider*, el sistema invoca el método en el mismo thread donde el cliente realizó la consulta. [**15**]. No existe llamada para destruir el proveedor. Cuando los datos son modificados a través de alguna llamada a los métodos *insert/update/delete*, la base de datos se abre y cierra atómicamente [**14**].

Cuando se leen datos llamando al mensaje *query*, debemos dejar la base de datos abierta. De lo contrario, los datos serán recolectados por el garbage collector (recolector de basura, en español).

Los *ContentProvider* continúan su ejecución a lo largo de la vida del proceso. Este proceso no es terminado a menos que el sistema operativo se encuentre con poca memoria entonces, si no está siendo utilizado por ninguna otra aplicación, podría llegar a ser destruido.

f

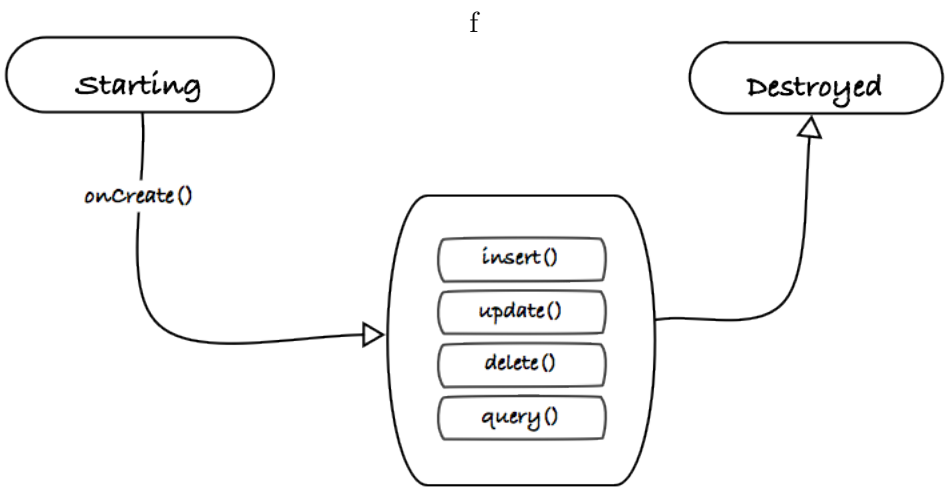


FIGURA 5. Ciclo de vida de un *ContentProvider*

Una aplicación cliente accede a los datos de un *ContentProvider* a través de un objeto *ContentResolver*. Este objeto provee distintos métodos como query(), insert(), update(), y delete() para acceder a los datos del *ContentProvider*. Luego, invoca métodos con nombres idénticos de una instancia de una subclase concreta de *ContentProvider*, que típicamente reside en un proceso separado. El *ContentProvider* actúa como una capa de abstracción entre los datos guardados y la presentación externa de mismos. Los objetos *ContentResolver* y *ContentProvider* manejan automáticamente los detalles de la comunicación inter-proceso.

2.3.2.2 Construyendo una consulta

El método ContentResolver.query() requiere varios argumentos:

- uri: la URI, usando el esquema content://.
- proyección: una lista de las columnas a devolver. Si este parámetro es nulo, devuelve todas las columnas, aunque esto es ineficiente.
- selección: un filtro donde se declara las filas a devolver en formato similar a una clausula WHERE de SQL excluyendo el WHERE. Si este parámetro es nulo, devuelve todas las filas para la URI dada.
- argumentos de selección: podrían incluirse símbolos de pregunta en la selección que serán reemplazados por los valores definidos en los argumentos de selección, en el orden en que aparecen en la selección.
- orden: cómo ordenar las filas en formato similar a la clausula ORDER BY de SQL excluyendo el ORDER BY. Si este parámetro es nulo, se usará el orden por defecto, que podría ser desordenado.

Este método devuelve siempre un *Cursor* conteniendo las columnas especificadas por la proyección de la consulta y las filas que cumplen con el criterio de selección de la misma.

Si un error interno ocurre, el resultado de la consulta dependerá del proveedor particular. Para algunos casos podría devolver un valor nulo, para otros lanzar una excepción. Algunas implementaciones de *Cursor* automáticamente actualizan al objeto cuando los datos del proveedor cambian, o lanzan un evento indicando que el *Cursor* cambió, o ambos.

2.3.2.3 Resumen del manejo de contactos en Android

Los proveedores de contenido de contactos del sistema Android manejan una base de datos extensible de información relacionada con contactos. Las aplicaciones Android pueden definir cuentas, que pueden almacenar información de contactos, incluyendo información a medida, en el proveedor de contactos del sistema. El proveedor de contactos del sistema engloba la información de todas las cuentas para luego exponer información de contactos unificada.

Por ejemplo, *Facebook*, *Twitter*, algunas aplicaciones de *Google*, y otras aplicaciones definen cuentas que almacenan información de contactos en el proveedor de contactos del sistema.

El proveedor engloba información común de distintas cuentas, de forma tal que un solo contacto Android pueda contener la dirección de email de una cuenta *Google*, el nombre de usuario de *Twitter*, números de teléfono de *Facebook*, etc.

2.3.2.4 Acceso al proveedor de contenido de contactos

La clase *ContactsContract*, algunas clases internas e interfaces relacionadas exponen acceso al proveedor de contactos del sistema. El proveedor de

contactos de Android maneja muchas tablas de información de contactos, incluyendo:

- ContactsContract.RawContacts: donde una fila representa un conjunto de datos que describen a una persona y la asocian con una única cuenta (por ejemplo, una de las cuentas de *Gmail* de un usuario).
- ContactsContract.Contacts: donde una fila representa un conglomerado de uno o más contactos de la tabla anterior presuntamente describiendo a la misma persona. Cuando los datos en la tabla RawContacts o datos asociados cambian, los contactos aglomerados afectados son actualizados de ser necesario.
- ContactsContract.Data: donde una fila almacena una única pieza de información de contacto (como por ejemplo el número de teléfono) y su meta-data asociada (como por ejemplo si es un número telefónico del trabajo o de la casa del contacto). El conjunto de tipos de dato que pueden ser almacenados en esta tabla se determina dinámicamente. Hay un conjunto predefinido de tipos comunes, definidos por las subclases de *ContactsContract.CommonDataKinds*, pero cualquier aplicación puede agregar sus propios tipos.

Notar que una aplicación que necesita acceso al proveedor de contenido de contactos necesita el permiso *android.permission.READ_CONTACTS* para realizar consultas y el permiso *android.permission.WRITE_CONTACTS* para insertar, actualizar o borrar datos de contactos.

2.3.3. Datos del dispositivo

Entre todos los datos del dispositivo que pueden ser de interés encontramos:

- El identificador único del dispositivo. Por ejemplo, el IMEI (del inglés International Mobile Equipment Identity, Identidad Internacional de Equipo Móvil) para GSM y el MEID o ESN para teléfonos CDMA. Este código identifica al aparato unívocamente a nivel mundial, y es transmitido por el aparato a una red en el momento en que se conecta a ésta. Esto quiere decir, entre otras cosas, que la operadora que usemos no sólo conoce quién y desde dónde realiza una llamada (utilizando la tarjeta SIM) sino también desde qué teléfono la hizo.

- El identificador único del abonado de un dispositivo. Por ejemplo, el IMSI (del inglés International Mobile Subscriber Identity, Identidad Internacional del Abonado a un Móvil) para un teléfono GSM. Éste es un código de identificación único del usuario del dispositivo móvil, integrado en la tarjeta SIM, que permite su identificación a través de las redes GSM y UMTS.
  Es usado para identificar al usuario de una red de celulares y es un identificador único asociado con todas las redes de celulares. Se guarda como un campo de 64 bits y es enviado por el teléfono a la red. También puede ser utilizado para obtener otros detalles del teléfono móvil. Para impedir que alguien externo identifique y monitoree al

abonado, el IMSI es enviado lo menos posible y se envía un TMSI (del inglés the Temporary Mobile Subscriber Identity, Identidad Temporaria del Abonado a un Móvil) generado de forma aleatoria en reemplazo.

- El número de serie de la tarjeta SIM (en inglés Subscriber Identity Module, Módulo de Identificación de Abonado). Ésta es una tarjeta inteligente extraible usada en teléfonos móviles y algunos módems que se conectan al puerto USB. Las tarjetas SIM almacenan de forma segura la clave de servicio del suscriptor usada para identificarse ante la red, de forma que sea posible cambiar la línea de un terminal a otro simplemente cambiando la tarjeta. El uso de la tarjeta SIM es obligatorio en las redes GSM.

- El número telefónico para la línea principal del dispositivo. Por ejemplo, el MSISDN (del inglés Mobile Station Integrated Services Digital Network, Estación Móvil de la Red Digital de Servicios Integrados) para un teléfono GSM. El mismo es también un número que identifica unívocamente un abonado en una red GSM o UMTS. Es el número telefónico de la tarjeta SIM en un teléfono móvil. El MSISDN junto con el IMSI son dos números importantes usados para identificar al abonado de un dispositivo. Este último identifica la tarjeta SIM, es decir, la tarjeta insertada dentro de los teléfonos celular mientras que el primero es usado para hacer llegar llamadas al abonado. El IMSI es comúnmente usado como clave en la base de datos que almacena la posición del usuario dentro de la red, mientras el MSISDN es el número normalmente marcado para conectar una llamada con el teléfono celular. Luego, la tarjeta SIM está unívocamente asociada a un IMSI, mientras el MSISDN puede cambiar, es decir, diferentes MSISDNs pueden estar asociadas a una tarjeta SIM, permitiendo a los usuarios ser llamados utilizando distintos números con un único dispositivo.

Un ejemplo de cómo pueden utilizarse estos datos para identificar a un usuario: Un usuario quiere hacer una llamada anónima, por lo que se compra un SIM en el tren y realiza una llamada usándolo. La llamada resultante no es anónima pues el usuario podría ser identificado utilizando el IMEI. Notar que los recursos como ser la línea de teléfono o el número serial de la tarjeta SIM son servicios que sólo tiene un teléfono celular y no otros dispositivos como ser una tablet.

Para acceder a alguno de los recursos listados, una aplicación debe tener en su manifiesto declarado el permiso *READ_PHONE_STATE*. Además de lo detallado anteriormente, este permiso puede ser utilizado para obtener información del estado actual del equipo, como por ejemplo, si se está realizando una llamada telefónica.

Un ejemplo típico de cómo una aplicación con permiso para acceder información de servicios de telefonía realiza una consulta:

```
TelephonyManager telephonyManager = (TelephonyManager)
    context.getSystemService(Context.TELEPHONY_SERVICE);
String id = telephonyManager.getDeviceId();
```

En este ejemplo, dado un contexto, obtenemos la instancia actual de *TelephonyManager* que nos provee de un servicio para obtener el identificador único del dispositivo, siendo el IMEI si se trata de una red GSM o el MEID para la red CDMA.

2.3.3.1 TelephonyManager

Entre los servicios que nos provee *TelephonyManager* encontramos varios muy interesantes que comprometen la privacidad de un usuario.
Teniendo en cuenta que varios de estos servicios permiten que un dispositivo y su dueño sean identificados unívocamente, consideramos de gran importancia controlar qué aplicación accede a este recurso.

En ASA logramos controlar el acceso a los siguientes servicios provistos por la clase *TelephonyManager*:

- getDeviceId: nos devuelve el identificador único del dispositivo (IMEI para red GSM o MEID para CDMA).
- getSubscriberId: nos devuelve el identificador único del abonado (IMSI).
- getSimSerialNumber: nos devuelve el número de serie de la tarjeta SIM.
- getLine1Number: nos devuelve un *String* con el número de teléfono de la línea principal del dispositivo.
- getNeighboringCellInfo: nos devuelve la información de las torres vecinas al dispositivo.
- getCellLocation: devuelve la ubicación actual del dispositivo (la torre a la cual se encuentra conectado).
- listen: registra un objeto que escucha notificaciones de cambios en un estado del teléfono en particular, entre los cuales se encuentran los listados más arriba.

2.3.4. Redes inalámbricas

En redes de computación, un conjunto de servicios (del inglés, Service Set) es un conjunto consistente de todos los dispositivos asociados con una red inalámbrica de área local (del inglés Wireless Local Area Network, WLAN). Un BSS (del inglés Basic Service Set, un conjunto básico de servicios) provee los pilares básicos para la construcción de una red 802.11 inalámbrica.

Entre todos los servicios que incluye el proveedor de datos de WiFi del dispositivo que pueden ser de interés encontramos:

- La dirección IP (del inglés Internet Protocol) del dispositivo. Una dirección IP es una etiqueta numérica que identifica, de manera lógica y jerárquica, a una interfaz (elemento de comunicación/conexión) de un dispositivo dentro de una red que utilice el protocolo IPv4, correspondiente al nivel de red del Modelo OSI. La dirección IP puede cambiar muy a menudo por cambios en la red o porque el dispositivo encargado dentro de la red de asignar las direcciones IP decide asignar otra IP al dispositivo (por ejemplo, con el protocolo DHCP).
- Dirección MAC (del inglés Media Access Control, Control de Acceso al Medio) del dispositivo. Es un identificador de 48 bits (6 bloques hexadecimales) que identifica de forma única a una tarjeta o dispositivo de red (es única para cada dispositivo). La misma contiene 24 bits que identifican al fabricante de la placa (para esto se utiliza el identificador único organizacional, del inglés Organizationally Unique Identifier). Las direcciones MAC son únicas a nivel mundial, puesto que son escritas directamente, en forma binaria, en el hardware en su momento de fabricación.
- El SSID (en inglés, Service Set Identifier) es el identificador de la red inalámbrica a la que se encuentra conectado el dispositivo y, que además, tienen todos aquellos dispositivos inalámbricos agregados a esa red.
- El BSSID (del inglés Basic Service Set Identifier) es el Identificador Básico del Servicio del punto de acceso actual. Cada BSS es identificado unívocamente por un BSSID. El BSSID es la dirección MAC del punto de acceso inalámbrico (en inglés, Wireless Access Point, WAP) generado combinando los 24 bits del identificador único del fabricante y los 24 bits asignados por el fabricante al chip del punto de acceso inalámbrico.
- Las configuraciones de redes WiFi del dispositivo. Estas configuraciones proveen información de las redes a las que el dispositivo se conectó alguna vez. Alguna de la información que provee de la red son SSID, BSSID, algoritmos de encriptación que utilizan y hasta las contraseñas almacenadas para cada red, entre otras cosas.
- Información de las redes a las que el dispositivo puede acceder en el momento. Estas redes son el producto del escaneo de la placa WiFi del dispositivo. Al igual que en el ítem anterior, los resultados del escaneo proveen información como SSID, BSSID, frecuencia de la red y algoritmos utilizados de encriptación de la misma.

Para acceder a alguno de los recursos listados, una aplicación debe tener en su manifiesto declarado el permiso *ACCESS_WIFI_STATE*.

Un ejemplo típico de cómo una aplicación con permiso para acceder servicios de redes inalámbricas realiza una consulta:

```
public String getIpAddr() {
    WifiManager wifiManager = (WifiManager) context.
        getSystemService(WIFI_SERVICE);
    WifiInfo wifiInfo = wifiManager.getConnectionInfo();
    int ip = wifiInfo.getIpAddress();
```

```
    String ipString = String.format(
                                "%d.%d.%d.%d",
                                (ip & 0xff),
                                (ip >> 8 & 0xff),
                                (ip >> 16 & 0xff),
                                (ip >> 24 & 0xff));

    return ipString;
}
```

En este ejemplo, dado un contexto, obtenemos la instancia actual de *WifiManager* que nos provee del servicio para obtener información de conexión y ésta tiene un servicio para obtener la dirección IP. Notar que este último nos devuelve un entero no formateado, por lo que podríamos, así como en el método del ejemplo, formatearlo a IPv4 canónico.

2.3.4.1 WifiManager y WifiInfo

Las dos clases principales modificadas de modo tal de controlar el acceso de aplicaciones al recurso de redes inalámbricas fueron el proveedor *WifiManager* y la clase *WifiInfo*. La primera nos provee de dos servicios importantes:

- getConfiguredNetworks: devuelve una lista de todas las redes configuradas. Entre los campos de cada red encontramos el identificador de la misma, su SSID y su BSSID, entre otros.
- getScanResults: devuelve una lista de los puntos de acceso encontrados en el escaneo de redes más reciente.

WifiInfo nos provee de otro conjunto de servicios que ASA también controla:

- getSSID: devuelve el identificador de la red inalámbrica actual, por ejemplo: *labosDC*
- getBSSID: devuelve la dirección MAC del punto de acceso actual.
- getIpAddress: devuelve la dirección IP del dispositivo.
- getMacAddress: devuelve la dirección MAC del dispositivo.

2.3.4.2 Otras formas de obtener información de WiFi

Finalmente, hay dos servicios adicionales, ambos en librerías internas de Android que permiten obtener la dirección IP del equipo. Ambos se encuentran en la librería *Utils* de Android: *getWiFiIpAddresses* y *getDefaultIPAddresses* que devuelven un *String* con la dirección IP formateada y separada por comas, si es que hay alguna, teniendo en cuenta direcciones del estilo IPv4 e IPv6.

2.3.5. Geolocalización

Como dijimos anteriormente, Android permite a las aplicaciones instaladas obtener la ubicación geográfica del dispositivo. Para ello provee una API cuya clase principal es la de *LocationManager*. Ésta es la encargada de

mediar los pedidos de las aplicaciones que intentan obtener información de la ubicación del dispositivo con los servicios del sistema que pueden proveer dicha información.

La API modela un conjunto de proveedores que obtienen la posición mediante distintos recursos de software y hardware del dispositivo, y se encargan de notificar al sistema de su posición.

Los proveedores utilizados por la mayoría de los dispositivos son *NetworkProvider* y *GPSProvider*. Estos proveedores se diferencian en la forma que tienen para obtener la ubicación, en el consumo de energía del sistema (un punto clave en los dispositivos móviles) y en qué tan precisos pueden ser. Ambos proveedores pueden ser deshabilitados por el usuario para evitar el consumo extra de batería desde la opción *Settings* ⇒ *Location Services* o desactivando el servicio *Google's Location Service* para detener al primero o *GPS Satellites* para el último.

Un detalle a tener en cuenta es que no todos los dispositivos Android tienen ambos proveedores, como es el caso de algunos que no tienen la posibilidad de obtener su posición a través de satélites de GPS por no tener el hardware necesario para ello.

Android provee dos permisos distintos para controlar el acceso a este recurso: *ACCESS_COARSE_LOCATION* y *ACCESS_FINE_LOCATION*. El primero permite obtener una ubicación aproximada a través del proveedor *NetworkProvider* usando información de las redes escaneadas y las torres de celulares, mientas que el segundo tiene una mayor exactitud de la posición del usuario a través del proveedor *GPSProvider* que utiliza satélites de GPS y mide su distancia a ellos. En caso de que una aplicación declare en el manifiesto el permiso de *ACCESS_FINE_LOCATION*, no es necesario que declare el otro para acceder a la información proveniente de las redes WiFi y torres.

Los métodos para obtener la ubicación del usuario que nos provee la API de Android se calculan a través de las redes WiFi, las torres de celulares o los satélites de GPS.

2.3.5.1 Torres de celulares [16]

Un celular está casi siempre conectado a la torre más próxima de modo de poder comunicarse con otros dispositivos a través de su operador de telefonía. La ubicación geográfica de cada torre es conocida por los operadores por lo que se les asigna el LAC (del inglés *Location Area Code*, Código de Área) correspondiente a esa ubicación. Estos datos nos permiten obtener una aproximación al lugar exacto donde se encuentra el dispositivo, pero que en muchos casos puede ser lo suficientemente buena. Es importante destacar que, a mayor cantidad de torres que compartan el mismo LAC, mejor

será dicha aproximación.

Notar que los identificadores de torres son un código que luego es utilizado para consultar la ubicación en forma de latitud y longitud.

Si volvemos a la sección de información de dispositivo y analizamos la API de *TelephonyManager* podemos observar que un dispositivo almacena esta información de la torre y puede ser consultada utilizando el servicio *getCellLocation*, aunque es necesaria una base de datos que mapee los identificadores a sus posiciones para poder calcular la ubicación del dispositivo.

Este método es usado a través de *NetworkProvider*. Su uso es preferible a los otros dos métodos respecto al consumo de batería, pues necesita de poco procesamiento, pero no lo es respecto a su precisión que puede llegar a diferir en miles de metros de la ubicación real [**18**].

2.3.5.2 Redes WiFi [13]

Un celular cuyo servicio de WiFi está prendido puede obtener la información de las redes inalámbricas que se encuentran a su alrededor. Android obtiene la ubicación del usuario a partir de estas redes escaneadas, gracias a un servicio provisto por *Google*. Esto es posible gracias a la recolección de información de las redes y la ubicación de un dispositivo de forma permanente en el momento en el que una persona habilita el servicio de *Google's Location Service*. De esta forma, el usuario acepta enviar información a *Google*, aparentemente anónima, con estos datos sin que necesariamente esté corriendo alguna aplicación.

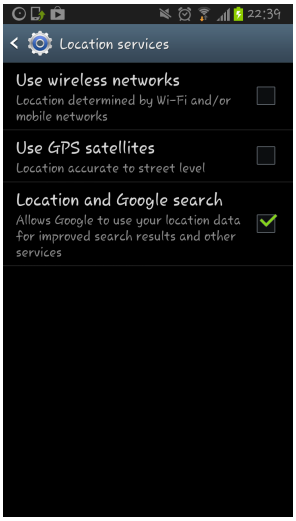


FIGURA 6. Habilitando Servicios de Ubicación de Google

Como se mencionó en previas secciones, la API de Android para obtener la información de las redes escaneadas es manejada a través de *WifiManager* utilizando el método *getScanResults*. Al igual que la ubicación utilizando las torres, se debe utilizar al proveedor *NetworkProvider*.

Su consumo de batería es mayor que al mecanismo anterior. Su exactitud depende de qué tantas redes WiFi hay alrededor, además de que las mismas hayan sido subidas y catalogadas por *Google*. El error respecto a la ubicación real podría llegar a los 200m, siendo este el alcance de una red WiFi.

2.3.5.3 Satélites GPS

Para poder calcular la ubicación del dispositivo, el mismo se conecta a varios satélites para poder determinar su distancia a cada uno. Cada distancia determina una esfera con referencia al satélite. Luego, es posible obtener la ubicación del dispositivo realizando la intersección entre estas esferas. A este método se lo llama trilateración.

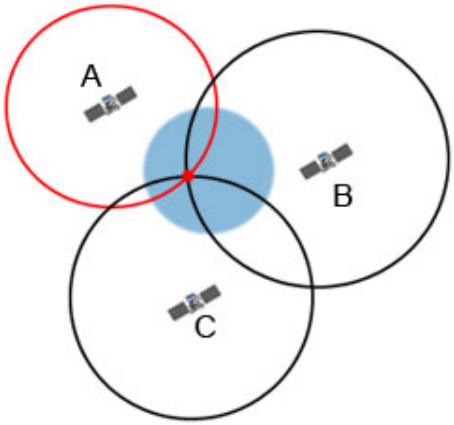


FIGURA 7. Trilateración

Esta fuente de ubicación es la que consume mayor energía. Sin embargo, no requiere acceso a internet, es la más precisa y puede ubicar a un usuario dentro de metros de su ubicación real. Su eficiencia no disminuye cuando el dispositivo está en movimiento mientras que la de los otros métodos sí. Las consultas de esta fuente se realizan con la API *LocationManager* utilizando el proveedor *GPSProvider*.

2.3.5.4 ¿Cómo obtiene la ubicación una aplicación?

Existen dos métodos que permiten a una aplicación obtener la ubicación de un dispositivo. Ambos utilizan la API de *LocationManager* y son independientes de la fuente.

El primero permite obtener la última ubicación almacenada de un proveedor utilizando el servicio *getLastKnownLocation*. Un detalle a tener en cuenta de dicho servicio es que puede devolver el objeto *null* si el proveedor nunca obtuvo una posición válida.

Otra forma es la de obtener actualizaciones periódicas de la ubicación a través de los servicios *requestUpdates* u obtener una única actualización al momento en que una ubicación válida esté disponible, a través del servicio *requestSingleUpdate*.

Hay dos objetos importantes cuando uno utiliza este tipo de mecanismos para obtener la ubicación actual. El primero es el *LocationListener* [**8**] que es una interfaz que debe ser implementada por la aplicación que quiere recibir noticias de que la ubicación actual cambió y es la responsable de recibir estas notificaciones, una vez que se haya registrado a través de los servicios anteriormente nombrados *requestUpdates* o *requestSingleUpdate*, según corresponda.

En el extracto de código que agregamos a continuación, suponemos que en la variable *locationManager* tenemos la instancia actual del *LocationManager* correspondiente. En este ejemplo, vemos una simple implementación de *LocationListener* que sólo registra la nueva ubicación detectada por el proveedor *NetworkProvider*.

Los métodos *onProviderDisabled*, *onProviderEnabled* y *onStatusChanged* fueron agregados por completitud dado que cualquier clase que extienda *LocationListener* debe implementarlos, pero no hacen nada dado que no son de interés para el ejemplo.

```
LocationListener onLocationChange = new LocationListener() {
        public void onLocationChanged(Location loc) {
                // Network provider found a new location
                Log.d("DEBUG", "New location: Lat>>" + loc.
                    getLatitude() + ", Long>>: " + loc.
                    getLongitude());
        }

        public void onProviderDisabled(String provider) {
                //not used
        }

        public void onProviderEnabled(String provider) {
                //not used
        }
```

```
        public void onStatusChanged(String provider, int status,
        Bundle extras) {
                //not used
        }
};

locationManager.requestLocationUpdates(LocationManager.
    NETWORK_PROVIDER,0,10000.0f, onLocationChange);
```

El segundo objeto que nos permite obtener la ubicación del dispositivo se llama *PendingIntent*. Éste permite que una aplicación pueda pasar a otra la capacidad de realizar operaciones que la primera definió con permisos propios. Podría entenderse como un token que es pasado a otra aplicación para que realice una acción sin importar si el proceso de la aplicación que otorgó el token está vivo o no. La aplicación que genera el token puede ser llamada utilizando un *BroadcastReceiver* (del inglés, receptor de transmisión) que, como su nombre bien lo indica, le permite recibir mensajes enviados por terceros o podría recibir un *Intent* utilizando el servicio *sendBroadcast*. También puede utilizarse para levantar una *Activity* que informe al usuario del cambio de su ubicación. Así como una aplicación puede escuchar notificaciones de cambio de ubicación del dispositivo a través del *LocationListener*, también podría registrarse a actualizaciones a través de una instancia de *PendingIntent*.

Capítulo 3

Prueba de concepto

Para implementar una prueba de concepto del sistema planteado, se tomó como referencia al recurso de contactos. Se utilizó una aplicación conocida por la mayoría de los usuarios de teléfonos inteligentes y utilizada por millones de personas en todo el mundo: *WhatsApp*.

WhatsApp Messenger es una aplicación de mensajería multiplataforma que te permite enviar y recibir mensajes sin pagar por SMS. *WhatsApp* está disponible para *iPhone*, *BlackBerry*, *Windows Phone*, *Android* y *Symbian*, y todos esos dispositivos pueden comunicarse del uno al otro. Debido a que *WhatsApp* usa el plan de datos que ya un usuario promedio tiene, no hay un coste adicional para enviar mensajes. La aplicación lee la agenda de contactos del dispositivo y, mantiene una base de datos de aquellos contactos para los cuales detecte que también usan la aplicación.

La implementación realizada durante la prueba de concepto nos permite controlar, para cada aplicación que acceda a la lista de contactos de nuestro dispositivo, qué grupos de contactos (los grupos de contactos definidos por Android) podrán ser accedidos por cada una de ellas. Esta modificación al control del recurso es totalmente transparente a la aplicación, de modo tal que, en este caso *WhatsApp*, no sólo no podrá tener acceso a los contactos que no pertenezcan a los grupos definidos en la configuración de ASA, sino que, además, no sabrá que existe una agenda con más contactos que aquellos a los que sí tiene acceso.

Como primer enfoque, probamos definir qué grupos serán visibles para *WhatsApp* de forma estática, fijados explícitamente en el código fuente al momento de modificar el acceso al proveedor de contactos. Dado que se trataba de una prueba de concepto, decidimos postergar la parte de la interfaz gráfica que permita al usuario configurar los grupos a los que cada aplicación que acceda al recurso de contactos tiene acceso, como se verá en la parte sección de arquitectura de nuestro sistema.

El objetivo de la implementación era que *WhatsApp* sólo pudiera tener acceso a los contactos pertenecientes al grupo de *Amigos* del dispositivo mientras que el resto de las aplicaciones debería tener acceso a todos los contactos.

Como explicamos en la sección anterior, al ser una aplicación de terceros, *WhatsApp* usa la API de Android para consultar la lista de contactos a través de *ContentResolver* llamando al método *query*. *ContentResolver*

obtiene al proveedor de contactos, llamado *ContactsProvider*, que es el encargado de finalmente resolver la consulta, a partir de una URI.

La primera idea que tuvimos fue la de alterar el comportamiento de *ContentResolver* y modificarlo en el momento en que se resuelven las consultas para aquellas que se refieran a contactos. El problema con esta idea es que interceptaríamos todas las consultas a *ContentResolver* perdiendo la abstracción de que se trata del proveedor de contactos.

Luego de analizar el código fuente y documentación oficial de Android, encontramos que toda consulta al proveedor de contactos termina en una consulta de más bajo nivel en la capa donde se encuentran las bases de datos *SQLite* del sistema, por lo que modificamos las consultas a *SQLite*. El problema con este nuevo enfoque es que a ese nivel perdíamos la abstracción del recurso (contactos en este caso). Al hacer esto, para modificar el acceso a la base de datos de contactos, estábamos alterando las consultas a todas las bases de datos del sistema. Esto no sólo puede afectar considerablemente la performance, sino que no tiene ventajas sobre nuestra implementación actual.

Una parte clave de modificar el comportamiento de la aplicación *WhatsApp* es la de detectar cuándo es esta aplicación la que requiere acceso a los contactos y cuándo no. Para esto, utilizamos una técnica algo precaria pero efectiva de lanzar una excepción y luego consultar la pila de ejecución para identificar qué aplicación fue la que originó la consulta.

Entonces, se alteró el método de las consultas de *SQLite* sobre aquellas llamadas que tenían en la pila de ejecución el nombre *WhatsApp*.

Se realizaron varias pruebas para determinar qué consultas realizaba *WhatsApp* al momento de obtener los contactos del sistema. Para eso utilizamos un log donde se guardan todos los accesos de la aplicación incluyendo a qué base de datos se realizaban. Entonces, encontramos que la misma realizaba muchas consultas a su base de datos propia *wa_contacts* que es una vista de la base */data/data/com.whatsapp/databases/wa.db*.

Incluso, dado que la mayoría de las consultas se realizaban a su propia base, parecía ser que no accedía a ninguna otra fuera de su propio espacio pero, luego de probar agregar y quitar contactos de la agenda de contactos del dispositivo, entre otras cosas se encontró que, al agregar un nuevo contacto, *WhatsApp* corrobora que las versiones de todos los contactos que tiene en su base estén actualizadas. Estas versiones se encuentran en las bases de datos que mantienen los contactos del dispositivo. En caso de encontrar una versión nueva de un contacto, *WhatsApp* realiza una consulta para obtener todos los contactos en el dispositivo (en particular contra la tabla en la ruta *com.android.contacts/data/phones*). Es por eso que, dicha consulta es la clave para modificar el acceso a los recursos por parte de esta aplicación.

Otra cosa que se notó es que las consultas de *WhatsApp* a las bases de datos del dispositivo no incluyen su nombre en la pila de ejecución. Se llegó a esto al observar que cuando se registran todas las consultas de *SQLite*, nunca vemos que aparezca el nombre de la aplicación, a pesar de que forzamos las consultas a la misma a través de la interfaz de *WhatsApp*.

Lo que ocurre es que *WhatsApp* utiliza la API de *ContentResolver* para obtener los contactos. Como se mencionó en conceptos previos, *ContentResolver* pasa la llamada al *ContentProvider* correspondiente quien resuelve la query creando un nuevo thread, dado que *WhatsApp* corre en un proceso distinto al *ContentProvider*. Es por esto que, una vez que imprimimos la pila de llamadas en el momento en que se realiza la query a *SQLite*, vemos la pila del nuevo thread creado por *ContentProvider* y perdimos el nombre de la aplicación que originalmente hizo la llamada al *ContentResolver*. El diagrama 8 muestra la interacción para obtener los contactos. El mismo es resultado de un análisis estático del código fuente de Android [**22**] y fue corroborado hookeando los distintos métodos mencionados en el diagrama para luego analizar la pila de ejecución en esos puntos.

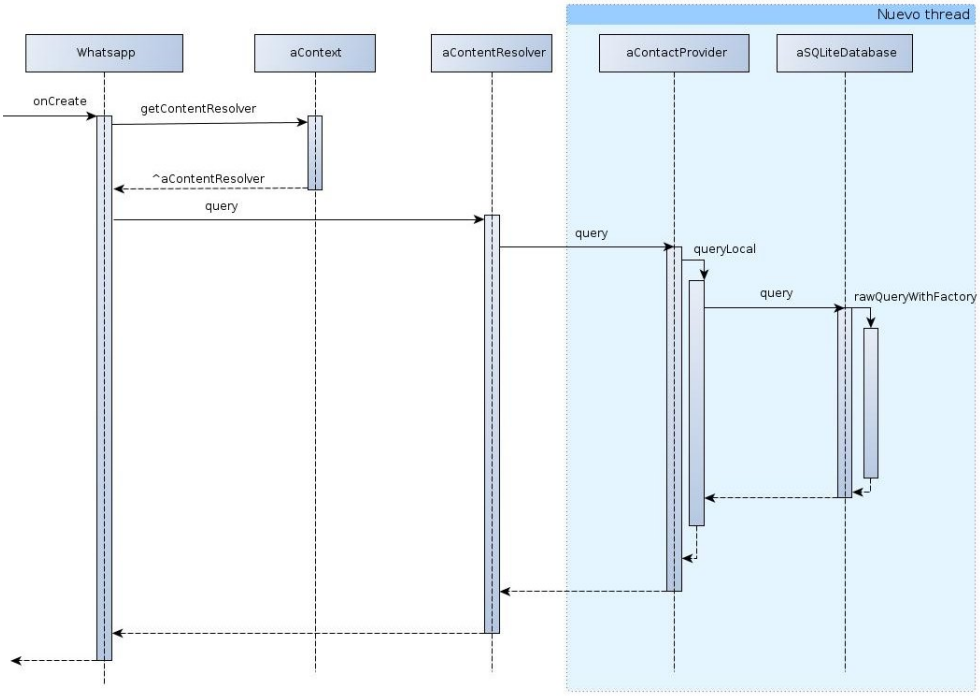


FIGURA 8. Comportamiento de *WhatsApp* al momento de consultar los contactos

Una vez que observamos este comportamiento, decidimos implementar un enfoque de más alto nivel: modificar las consultas de la API de *ContentResolver* y alterar la selección de las queries a cualquiera de las

bases de datos de contacto. Como ya dijimos, a este nivel se puede identificar correctamente cuál es la aplicación que realizó originalmente la consulta.

Como mencionamos anteriormente, la clave es modificar la consulta de la aplicación a la tabla en *com.android.contacts/data/phones*, identificada con la URI *ContactsContract.CommonDataKinds.Phone*. Finalmente, modificamos el método de consulta de *ContentResolver* para, cuando se trata de una consulta de la aplicación seleccionada hacia dicha tabla, se filtren aquellos contactos que no pertenecen a los grupos especificados.

Entonces, la implementación de dicha alteración lleva a que los resultados del cursor devuelto por la consulta coincidan con los contactos que pertenecen al grupo *Amigos* y ninguno más.

Para corroborar que la aplicación no tiene acceso a los demás contactos del dispositivo se utilizó la herramienta *SQLite Manager Plugin* explicada en el apéndice. Con ella se pudo corroborar que la base de datos interna de *WhatsApp* sólo tiene aquellos contactos pertenecientes al grupo especificado.

3.1. Conclusiones de la prueba de concepto

De la implementación de la prueba de concepto de administración de contactos rescatamos algunas observaciones importantes:

- WhatsApp realiza consultas a una de las tablas del sistema que contiene contactos cada vez que detecta que su base de datos local de contactos está desactualizada.
- Sabemos que todas las bases de datos que manejan contactos son administradas por el proveedor de contenido de contactos llamado *ContactsProvider*.
- Si queremos utilizar la técnica de identificación de aplicaciones observando la pila de ejecución, debemos modificar las llamadas a la API de *ContentResolver* y no al *ContentProvider* ni las consultas sobre las bases de datos de *SQLite*.
- Al modificar las queries realizadas a nivel de la clase *SQLiteDatabase* no sólo se modifican los accesos a una base de datos particular sino a todas las bases de datos del sistemas, lo cual perjudica la performance del mismo, además de ser un enfoque demasiado invasivo.

Capítulo 4

Arquitectura ASA

4.1. General

Para poder implementar el sistema planteado, modificamos algunos comportamientos puntuales de la API de Android, en particular sólo modificamos los proveedores de recursos con los que las aplicaciones se comunican al momento de acceder a ellos. Ésta vendría a ser la capa que está justo debajo de la capa donde se encuentran las aplicaciones.

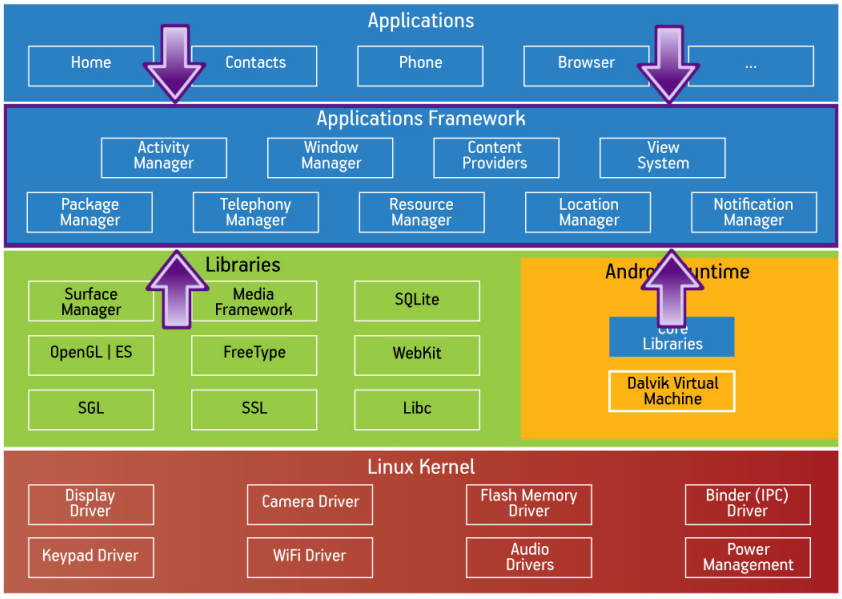


FIGURA 9. Capas de Android

En la figura 10, mostramos un pequeño pantallazo de nuestra arquitectura.

Aquí se ven con fondo liso los componentes de Android, con rayas diagonales los nuestros y con rayas horizontales los componentes de Android que modificamos.

Originalmente, cuando una aplicación quiere acceder a un recurso, se verifica en el sistema de permisos si dicha aplicación tiene el permiso necesario y luego, se envía el pedido al proveedor del recurso que finalmente accede a

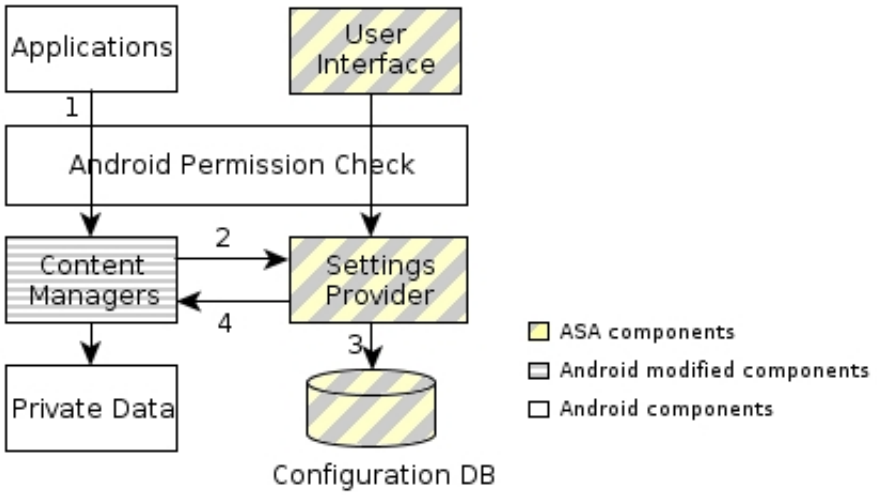


FIGURA 10. Arquitectura ASA

los datos y se los devuelve a la aplicación.

Nuestra modificación *hookea* esta llamada. Antes de realizar la consulta de los datos reales, identificamos la aplicación que está intentando acceder al recurso. Luego, buscamos la política a aplicar definida por el usuario para esa aplicación y ese recurso.
Una vez obtenida la configuración, modificamos la consulta original de forma totalmente transparente para la aplicación que está queriendo acceder al recurso. De esta forma, según está definido por la política, se devolverá a la aplicación un subconjunto del resultado original o bien datos ficticios.

Además de los módulos mencionados, contamos con una interfaz de usuario que permite definir las políticas para cada recurso y aplicación de terceros o del sistema que lo acceda.

Antes de ahondar en la arquitectura de cada recurso, es necesario conocer qué aspectos de cada uno de ellos son administrados por ASA. La administración del recurso de contactos se basa en el sistema de grupos de Android. Un usuario puede definir una política de acceso de agenda de contactos de una aplicación determinando cuales son los grupos a los que la misma puede acceder.

Por otro lado, la política de información del dispositivo (IMEI, IMSI, etc) permite al usuario seleccionar entre tres posibles mecanismos que se utilizarán para definir el tratamiento de cada valor. Para la información de IMEI, IMSI, número de SIM y número de linea se podrá definir si una aplicación puede acceder al valor verdadero, uno falso o uno aleatorio cada vez que sea consultado. Además, el usuario puede editar estos datos usando valores a medida, siempre y cuando cumplan con el formato esperado. El resto de los

servicios controlados por ASA que fueron mencionados en la sección donde describimos los servicios tomados en cuenta de *TelephonyManager* 2.3.3.1 son utilizados por el módulo de geolocalización al modificar la ubicación con la que queremos mentirle a una aplicación.

Para administrar la información proveniente de las redes WiFi el usuario puede definir el acceso a la información mencionada en la sección donde describimos los servicios de *WifiManager* y *WifiInfo* 2.3.4.1, devolviendo información falsa o la real pero con el agregado extra de poder seleccionar una red WiFi guardada previamente como política a aplicar. De esta forma un usuario podría guardase la configuración de una red WiFi cualquiera y usarla en otro momento para configurar una aplicación dada. Por ejemplo, podría guardarse la información de red de la oficina donde trabaja y luego usarla desde la casa. Entonces, si una aplicación a la que se le aplicó una política del estilo preguntase por la SSID o BSSID, obtendría aquellos valores de la WiFi guardada. Además, un usuario puede modificar los valores de la WiFi guardada y definir una política tan fina como quiera, siempre y cuando los nuevos valores sean válidos.

Por último se encuentra el módulo de geolocalización. Un usuario puede definir uno de tres valores posibles: permitir que una la aplicación acceda a la ubicación real del dispositivo, seleccionar una ubicación en un mapa de *Google Maps* para engañar a una aplicación cuando pregunte la ubicación al sistema de GPS o seleccionar una de las ubicaciones almacenadas previamente, de forma similar a la descripta para el recurso de WiFi. Notar que, en el caso en que un usuario marque una ubicación no guardada por él en el mapa, ASA fuerza a que la aplicación configurada, para usar dicha ubicación, sólo use el recurso de GPS devolviendo una celda nula en el caso de las torres de celulares y una lista vacía en el de las redes escaneadas.

4.1.1. El problema del contexto

Con el problema del contexto nos referimos a un detalle implementativo de Android que obliga a un desarrollador a, casi en todo momento, tener un objeto que representa un contexto para poder usar ciertos servicios que de alguna forma son indispensables para una aplicación.

La clase *Context* de Android está definida como una interfaz a información global del ambiente de una aplicación. Permite acceder a recursos y clases específicas de una aplicación, así como operaciones a nivel de aplicación como la de lanzar una actividad, intents y demás estructuras de principal importancia para poder implementar el flujo de una típica aplicación Android.

En el caso de ASA en particular, necesitamos de un contexto para poder instanciar lo que se denomina un *helper* de una base de datos *SQLite* de forma tal de poder acceder a las configuraciones del usuario en nuestro sistema.

Notar que, en el momento en que estemos extendiendo código de la API de Android con comportamiento propio, vamos a estar corriendo con los permisos de la aplicación que originalmente hizo la llamada y en el ambiente de dicha aplicación, no en el de nuestro sistema. Es por eso que debemos acceder a nuestra propia base de datos pasando por los controles de Android dado que esa base de datos no está en el espacio de la aplicación controlada, sino en el de ASA.

Otra razón por la que necesitamos un contexto es para identificar la aplicación que está realizando la llamada y decidir si la misma es del sistema o es una aplicación de terceros. Sin entrar mucho en detalle, para poder hacer esto necesitamos instanciar un *PackageManager* que es un objeto de Android que contiene información de todas las aplicaciones en el dispositivo y, para obtener una instancia del mismo, necesitamos un contexto. Este nos permite decir si se trataba de una aplicación preinstalada o no y además nos permite obtener los paquetes de software asociados a un UID, cosa útil que explicaremos más adelante.

La buena noticia es que logramos obtener un contexto en todo momento. Cada aplicación, mientras está corriendo, tiene una copia de las librerías que la misma utiliza [**1**]. En cada instancia de estas librerías ASA almacena, en una clase propia, una referencia al contexto de dicha aplicación. Notaremos que, si tenemos una referencia a un contexto para una aplicación, podemos estar tranquilos de que dicho contexto es útil para instanciar una base de datos o un *PackageManager*. Es importante destacar que el contexto no es utilizado para identificar a la aplicación en cuestión dado a que esto podría llevar a problemas de *TOCTOU* (Time of check to time of use).

Para obtener un contexto, nos valemos de las herramientas que nos provee Cydia Substrate y, modificamos algunos servicios de la clase *Context* que, de alguna forma, estamos seguros serán llamados por la aplicación de interés o bien simplemente usamos *Java Reflection*. En el primer caso, al momento en que se llama al servicio, nos guardamos el contexto actual. El segundo lo utilizamos cuando sabemos que existe un método de la clase modificada que nos permite acceder a un contexto válido, como es el caso de la clase *Phone* que tiene un método público *getContext* pero no accesible desde la API de Android.

4.1.2. Cómo identificamos una aplicación

Un detalle a tener en cuenta es que debemos detectar qué aplicación está intentando acceder al recurso cuyo acceso estamos controlando, en tiempo de ejecución. Por ejemplo, dado una llamada a *ContactsProvider*, el proveedor de contenido de contactos, queremos identificar la aplicación que inició la consulta, para decidir qué configuración aplicar al momento de devolver los resultados de la misma.

4.1.2.1 Usar la pila de ejecución

Poco eficiente pero eficaz, como primer intento decidimos lanzar una excepción e inspeccionar la pila de ejecución. De la pila podemos obtener la aplicación que realizó la consulta y buscar la configuración a aplicar mediante una nueva consulta a nuestra base de configuraciones. Luego, se aplica la configuración devolviendo un subconjunto del resultado original en vez del original en sí. Este método se pudo utilizar siempre y cuando no se haya creado un nuevo thread desde que la aplicación que queremos identificar hizo originalmente la llamada, como detallamos anteriormente. Es por eso que se utilizó cuando nuestra implementación modificaba a nivel de la clase *ContentResolver*.

Código para obtener el stack trace:

```
String stack trace = "";
try{
        throw new Exception("DEBUG");
}
catch(Exception e){
        stack trace = Log.getStackTraceString(e);
}
```

Una vez que tenemos la pila de ejecución, listamos todas las aplicaciones instaladas en el sistema y buscamos alguna que se encuentre en la pila de ejecución. Este primer intento detectaba de forma exitosa la aplicación que realizó la consulta a *ContentResolver* pero era muy poco eficiente. Por lo que, antes de realizar una búsqueda entre todas las aplicaciones, filtramos las que no tienen acceso al contenido al que se quiere acceder. De esta forma se redujo mucho la cantidad de aplicaciones para las cuales se verificaba si estaban en la pila, aunque sabemos que la consulta de los permisos por cada aplicación sigue siendo poco eficiente.

El volver a obtener las aplicaciones instaladas, filtrarlas, etcétera por cada nueva consulta al *ContentResolver* terminaba siendo un problema.

Finalmente, surgió la idea de sólo buscar en la pila de ejecución, las aplicaciones para las que el usuario configuró el acceso al recurso controlado, reduciendo bastante el espacio de búsqueda para algunas instancias.

Cuando implementamos la estrategia de identificación pudimos encontrar dos problemas: por un lado, cuando examinamos el contenido del stack trace en la llamada de query al *ContentResolver* se puede encontrar el nombre del paquete del originario de la llamada. En cambio, cuando lo hacemos directamente sobre las consultas de la base utilizando la clase *SQLiteDatabase* o sobre el *ContactsProvider*, modificando alguna de las formas de consultas (si bien la clase tiene muchos métodos para consulta, todos terminan utilizando el mismo método *rawQueryWithFactory* para *SQliteDatabase* y *query* para *ContactsProvider*) lo que ocurre es que no se tiene forma de identificar a la aplicación originaria de la query a través del

stack trace, pues la consulta se realiza en un nuevo thread con un stack trace diferente al que inició la consulta.

La segunda debilidad se basa en el hecho que el nombre de las aplicaciones incluido en el stack trace no siempre coincide con el nombre del paquete (aplicación) sino que incluye el nombre del proceso. Un ejemplo a mencionar es el de la aplicación *DW Contacts & Phone & Dialer* donde el nombre del proceso es *dw.android.contacts* mientras que el nombre del paquete *dw.android.contacts.free* y en stack trace sólo se menciona el primero.

4.1.2.2 Identificación utilizando PID

Otra estrategia para identificar una aplicación en tiempo de ejecución es la de utilizar algunos servicios provistos por el sistema de archivos de Linux. Inspeccionando el directorio /proc, que es un sistema de archivos virtual montado en el directorio raíz encontramos información de cada uno de los procesos que se encuentran corriendo en el sistema, como ser su nombre, sus variables de entorno, el directorio donde se encuentra su ejecutable, información del mapeo de su memoria, entre otras cosas. Luego, podemos obtener el nombre del proceso que está corriendo utilizando su PID (identificador de proceso) consultando al archivo que se encuentra en */proc/<pid>/cmdline*.

Para obtener el PID o el UID del proceso que esta accediendo al recurso podemos usar dos servicios de la API de Android:

```
int uid = Binder.getCallingUid();
int pid = Binder.getCallingPid();
```

Luego, la siguiente implementación obtendría cuál es el nombre del proceso que está corriendo actualmente:

```
public static String getProcessNameByPid(int pid) {

        BufferedReader bufferReader;
        try {
                bufferReader = new BufferedReader(new FileReader
                    (("/proc/" + pid + "/cmdline")));
                String processName = bufferReader.readLine();

                 bufferReader.close();
                return processName;

        } catch (Exception e) {
                Log.d("ERROR","There was an error getting the
                    pid");
        }

        return null;
}
```

Identificar una aplicación usando sólo su PID no es suficiente pues dos aplicaciones distintas podrían tener el mismo nombre de proceso. Luego de leer la documentación de Android, encontramos que una aplicación puede

definir el nombre de proceso que usará en su archivo manifiesto bajo la propiedad *android:process*. Se realizó una prueba que confirmó que nuestra aplicación puede definir su nombre de proceso como, por ejemplo, el de alguna aplicación del sistema. Tampoco es suficiente usar sólo el identificador de usuario: Android le asigna a cada aplicación un nuevo UID a menos que ellas indiquen que quieren compartirlo con otras aplicaciones. En este caso, ambas aplicaciones deben estar firmadas por un mismo autor. El problema está en que en el caso que las aplicaciones compartan el UID, no hay forma de determinar qué regla aplicar a partir del nombre del paquete que se encuentra en las bases de datos de las preferencias de ASA. Esto justifica que usemos el nombre de proceso en combinación con el UID de la aplicación para determinar la política a aplicar.

Consultamos por mail a los autores del sistema de TISSA [**5**], presentado en la sección de *Trabajos relacionados*, sobre cómo resolvían este problema. Nos respondieron que TISSA identifica las aplicaciones sólo mediante el UID y supone que las mismas no lo comparten. Nosotros quisimos dar un paso más. Por eso, para decidir si dada una política, ésta debe ser aplicada sobre el acceso de una aplicación al recurso controlado, chequeamos que el nombre del paquete definido en la política esté incluido entre los posibles paquetes que comparten el UID de la aplicación y, además, que el proceso asociado al PID de la aplicación esté incluido en la lista de procesos del paquete al cual la política se refiere. Esto nos permite, en el caso que las aplicaciones compartan el UID, considerar la distinción de las mismas mediante su nombre del proceso.

Es importante aclarar que, cuando un conjunto de aplicaciones comparten el mismo UID y tienen los mismos nombres de proceso, ASA aplicará la primera de las políticas de dicho conjunto.

Cuando se configura la política de una aplicación, para que empiece a regir desde ese momento, es necesario terminar los procesos que esa aplicación está corriendo. De otra forma, aquellos datos almacenados en variables internas pueden almacenar todavía la información privada que se estaba intentando proteger. Si bien terminar la aplicación podría perjudicar la usabilidad, esta es una decisión de diseño tomada en ASA. Otra opción sería la de avisar al usuario que es necesario reiniciar el equipo para que la política tenga efecto. Sin embargo, la decisión de terminar los procesos involucrados, sólo afecta a la aplicación sobre la cual se acaba de modificar la política de control de acceso y se le avisa al usuario de dicho efecto.

4.2. Componentes principales del sistema

4.2.1. Interfaz de usuario y funcionalidad adicional

ASA provee una interfaz de usuario simple para mejorar su usabilidad. Se tiene una *Activity* por cada recurso considerado en donde se listan las aplicaciones instaladas que utilizan ese recurso. Además, ASA separa el listado de aquellas aplicaciones provenientes del sistema (aquellas pre-instaladas) de las de terceros.

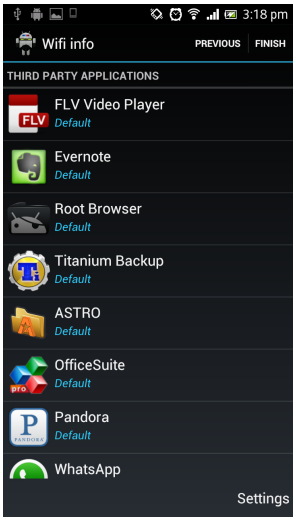


FIGURA 11. Listado de Aplicaciones que leen información de WiFi

Cuando se selecciona una aplicación de la lista, se crea una nueva *Activity* en donde se puede configurar la política a aplicar. La misma depende del recurso seleccionado.

En el caso del recurso de contactos, se listan los grupos de contactos definidos en el dispositivo, como se muestra en la figura 12. Debido a que puede haber grupos provenientes de distintas cuentas, podríamos tener dos grupos con el mismo nombre. Por ejemplo, podríamos encontrarnos con el grupo de Amigos de mi cuenta de *Gmail* y el grupo Amigos almacenado en el dispositivo, por lo que la lista de grupos de contactos está a su vez agrupada según la cuenta a la que pertenecen. Notar que los grupos del dispositivo se agregan en una cuenta aparte.

La interfaz definida para configurar la política de información del dispositivo, que se muestra en la figura 13, permite al usuario seleccionar entre tres posibles mecanismos que se utilizarán para definir el comportamiento de cada valor. Éstos pueden ser definidos como *Random*, *Fake* o *Real*.
Además, como funcionalidad adicional, ASA permite al usuario definir una política a medida que llamamos *Custom*. Como se puede observar en la figura 14, un usuario puede editar los valores de cada valor controlado por

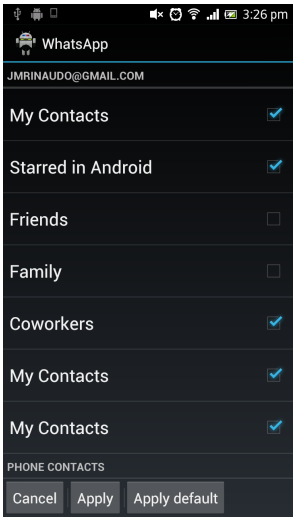


FIGURA 12. Definiendo política de contactos para WhatsApp

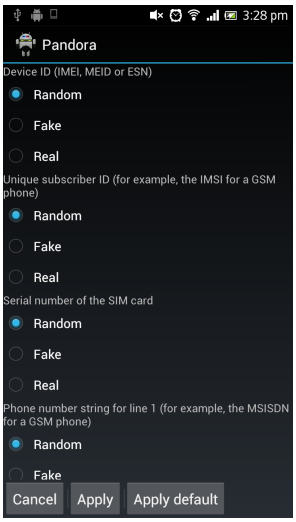
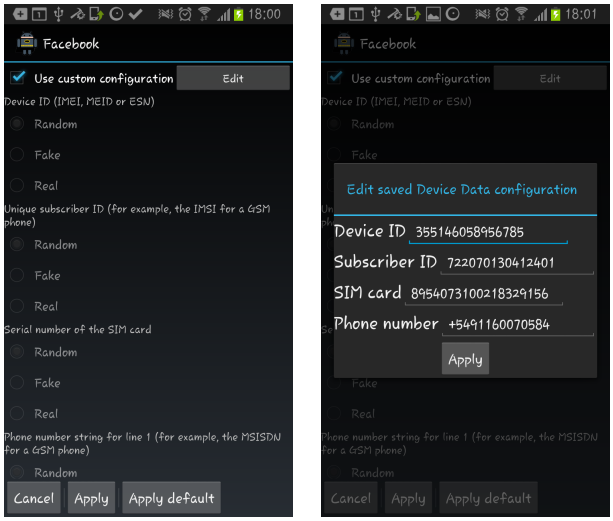


FIGURA 13. Definiendo política de datos del dispositivo para Pandora

ASA, si así lo desea. Para esto, una vez que el usuario tilda la opción de definir datos del teléfono *Custom*, otro diálogo muestra los valores reales de su dispositivo para poder editarlos según lo necesite.

Para el módulo de las redes WiFi, la interfaz lista la información proveniente de una red inalámbrica y se le puede asignar las opciones *Fake* o *Real* para cada una. Como se puede observar en la figura 16, un diálogo permite seleccionar una de las WiFis guardadas para utilizar como política de forma que las aplicaciones al acceder obtengan esos valores definidos por dicha red. Adicionalmente, un usuario puede editar los valores de la red guardada, si así lo desea.



(a) Opción para definir datos del teléfono *Custom*

(b) Editando los valores de teléfono para *Facebook*

FIGURA 14. Funcionalidades especiales de la GUI de ASA

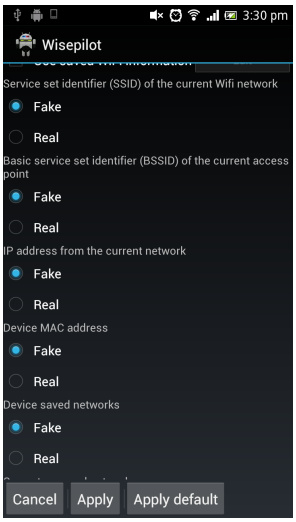
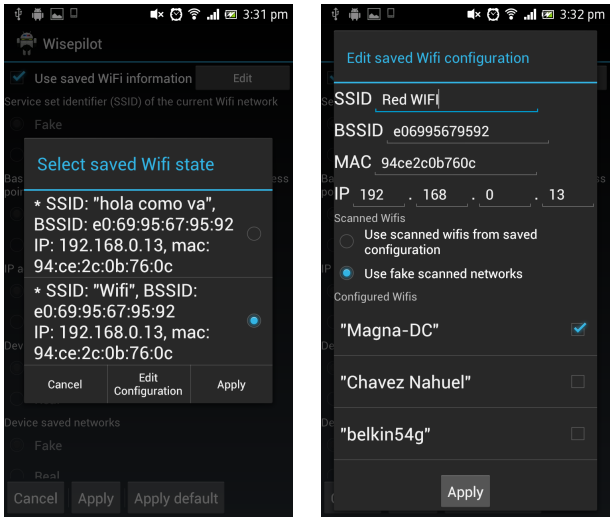


FIGURA 15. Definiendo política de *redes inalámbricas* para *Wisepilot*

Por último, ASA provee una interfaz para definir una política del recurso de geolocalización. La misma contiene un mapa de *Google Maps* para seleccionar alguna de las ubicaciones guardadas o simplemente un punto en ese mapa que se define como una nueva ubicación, forzando el uso de la geolocalización mediante GPS.

Para agregar tanto una configuración de WiFi como de ubicación se debe ir a la ventana de configuración mostrada en la figura 18, que se encuentra en la ventana respectiva para cada recurso.



(a) Usando un estado de WiFi guardado

(b) Editando los valores de una WiFi guardada

FIGURA 16. Funcionalidades especiales de la GUI de ASA

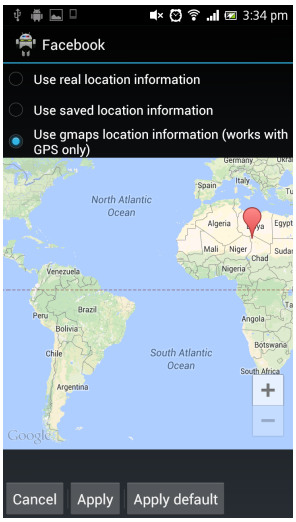


FIGURA 17. Definiendo política de ubicación para *Facebook*

4.2.2. Instalación de ASA

Durante la instalación del sistema se inicia un asistente destinado a ayudar al usuario a configurar las preferencias generales iniciales de ASA. La figura 19 muestra una instancia de este asistente.

Hay tres configuraciones generales distintas que denominamos modos: un modo paranoico, para los usuarios que prefieren un control completo al acceso de su información privada, un modo seguro para usuarios que confían

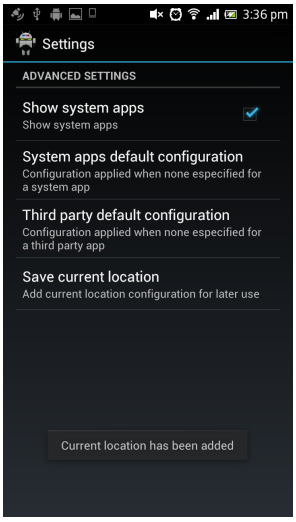


FIGURA 18. Guardando ubicación actual

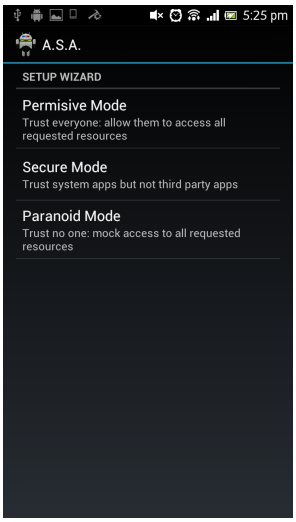


FIGURA 19. Asistente de instalación

en las aplicaciones del sistema pero no en las de terceros, y un modo permisivo, para usuarios que en un principio no esperan gran impacto sobre la administración de los recursos de su dispositivo.

Mientras que el modo permisivo permite instalar la aplicación sin generar ningún impacto sobre el comportamiento de las aplicaciones existentes, el modo paranoico modifica el comportamiento de las aplicaciones por accesos más restrictivos o anónimos. Definiendo un módulo para cada recurso, ASA decide cuál será su comportamiento inicial por defecto, ya sea para aplicaciones del sistema o para aplicaciones de terceros.

Por ejemplo, para geolocalización, el modo paranoico implica que tanto aplicaciones del sistema como de terceros obtienen una ubicación falsa predefinida por ASA. En cambio, el modo seguro permite a las aplicaciones del sistema acceder a la ubicación real del dispositivo mientras que sigue mintiéndole a aplicaciones de terceros.

Para el subsistema encargado de manejar el acceso a contactos en modo permisivo, se permite el acceso a todos los grupos del dispositivo para cualquier tipo de aplicación, el modo seguro no expone la presencia de ningún contacto dentro de ningún grupo para aplicaciones de terceros pero exhibe todos los contactos para aplicaciones del sistema. El modo paranoico oculta todos los contactos para cualquier tipo de aplicación.

Una buena práctica es que ASA sea la primera aplicación instalada en el sistema para disminuir la probabilidad de una posible divulgación de información sensible proveniente de cualquier tipo de recurso controlado por nuestro sistema.

4.2.3. Configuración por defecto

Una funcionalidad interesante que favorece la usabilidad y la performance es el de considerar reglas predeterminadas para las aplicaciones en caso de que el usuario no haya definido una política determinada para una aplicación y un recurso dados.

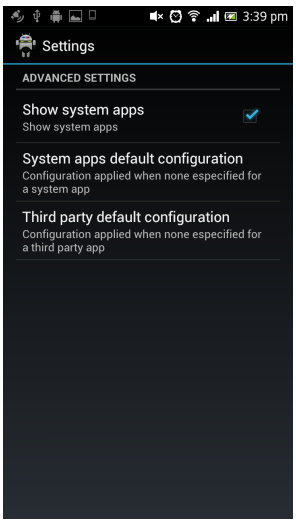


FIGURA 20. Interfaz para definir política por defecto en Contactos

Esta funcionalidad permite al usuario definir una regla para aquellas aplicaciones cuyas configuraciones no fueron definidas particularmente. Esta fucionalidad que se presenta en la figura 20, se permite definir una política

general para aplicaciones de terceros y otra para aplicaciones del sistema.

Al momento en que ocurre una llamada a uno de los métodos modificados, se identifica a la aplicación que está queriendo acceder al recurso en cuestión y luego se busca una configuración definida por el usuario a aplicar para la misma. Si no encuentra ninguna configuración para aplicar, se consulta al proveedor de configuraciones nuevamente para obtener el comportamiento por defecto, según la aplicación sea de terceros o del sistema.

Un usuario tiene la posibilidad de resetear la configuración de una aplicación a la configuración default. En ese momento, su configuración particular se elimina de la base de datos de configuraciones del recurso considerado.

Un detalle a tener en cuenta es que se tomó la decisión de catalogar a una aplicación como del sistema cuando alguno de todos los paquetes que comparten ese mismo UID son paquetes instalados por defecto en el dispositivo, dado que al momento en que se debe verificar si se trata de una aplicación del sistema o no, tenemos únicamente acceso al PID y al UID de quien se encuentra realizando la llamada.

Un enfoque más restrictivo podría haber sido el de considerar que una aplicación es del sistema sólo cuando todos los paquetes instalados de ese UID lo son. El problema con esta decisión es que aplicaciones que uno intuitivamente considera como del sistema, terminan no siéndolo. Un ejemplo puntual es el de la aplicación de contactos que viene preinstalada en el dispositivo, que no es considerada del sistema porque una aplicación llamada *Google Services* comparte su UID pero no vino originalmente con el dispositivo. Lo mismo ocurre para otras aplicaciones provistas por *Google* y que vienen preinstaladas en el equipo. En un futuro se podría agregar que un usuario defina perfiles, formados por un conjunto de políticas, de forma tal de poder asignar un perfil a una aplicación.

Una funcionalidad importante de ASA es tener un comportamiento por defecto que permite que un usuario pueda instalar una aplicación del mercado sin tener ninguna preocupación de qué información sensible obtuvo la aplicación antes de que haya tenido un momento para configurar sus preferencias de acceso a recursos en ASA. Es decir, las aplicaciones son controladas por ASA desde el momento en que son instaladas en el dispositivo (siempre y cuando ASA haya sido instalada primero). Esto se distingue de otros trabajos relacionados como TISSA [**5**].

Si bien la modificación de la política por defecto de algún recurso se efectiviza en el instante en que el usuario las realiza, puede ocurrir que aplicaciones que estaban corriendo tengan almacenada información privada que la política anterior no le restringía. Esto se debe a que, a diferencia de cuando cambiamos la política de una aplicación particular, en ASA se decidió no terminar a todas las aplicaciones que acceden a un recurso dado cuando su configuración por defecto es modificada, para evitar afectar la estabilidad

del sistema.

Sin embargo, para el próximo acceso al recurso por parte de dichas aplicaciones, la nueva política será aplicada. Dado que se decidió no terminar a todos aquellos procesos que tengan almacenada información sensible luego de un cambio de una configuración general, se recomienda al usuario que considere reiniciar el equipo para forzar a que todas las aplicaciones deban consultar nuevamente a los recursos y la nueva política sea efectivamente aplicada.

4.2.4. Agenda de contactos

4.2.4.1 Granularidad

Definimos la granularidad de contactos en base al sistema de grupos de Android. De esta forma, un usuario puede definir un rol distinto para cada aplicación que acceda a los contactos del dispositivo de modo tal que podría decidir dar sólo acceso a aplicaciones que usa en el ámbito laboral, como puede ser, por ejemplo, *LinkedIn* para que sólo acceda al grupo de *Trabajo* y, en cambio, definir que una aplicación como *WhatsApp* sólo tenga acceso a los contactos en los grupos de *Familia* y *Amigos* porque, por ejemplo, no tiene interés que *WhatsApp* sepa que tengo a mi jefe en mi agenda de contactos.

Como ya dijimos anteriormente, esta decisión mejora la usabilidad, pero además le da un mayor poder de control al usuario.

4.2.4.2 Implementación de control de acceso a contactos

La idea del sistema desarrollado es intentar controlar el acceso de aquellas aplicaciones que realizan lectura de los contactos del dispositivo. Para ello, es necesario analizar la documentación de la API de Android y ubicar las posibles formas que las aplicaciones tienen para leer los contactos. Se utilizó una tablet donde probar nuestro sistema cuya versión de Android es *4.0.3* . Dado que el framework de Cydia Substrate está disponible a partir de la versión *2.3* , analizamos la documentación disponible de la API entre dichas versiones.
De dicho análisis surge que existen distintas consultas que pueden ser realizadas a varias bases de datos *SQLite* del sistema, las cuales siempre nos devuelven una instancia de la clase *Cursor* que es una clase encargada de proveer un acceso de lectura-escritura al conjunto de resultados obtenidos de una consulta a una base de datos.

Como puede observarse en el diagrama 21, una aplicación puede realizar dichas consultas utilizando la API de Android de alguna de las siguientes formas:

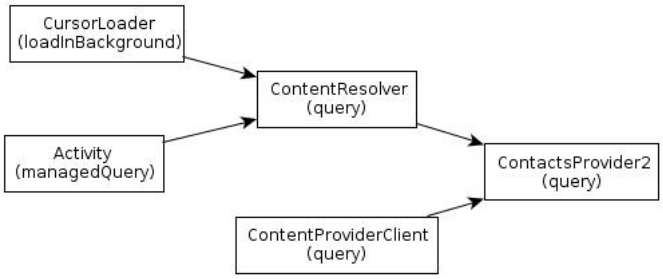


FIGURA 21. Consultas al recurso de contactos

- Utilizando la clase *CursorLoader*, cuyo método *loadInBackground* devuelve un *Cursor*.
- A través de la clase *Contacts.People*, deprecada a partir de la API nivel 5, utilizando su método *queryGroups* que utilizaba el proveedor de contactos *ContactsProvider*.
- Desde la API nivel 5 en adelante, Android provee la clase *ContactsContract* que es una nueva abstracción de las bases de datos de contactos. El proveedor correspondiente con el cual una aplicación se comunica para acceder a esta información es *ContactsProvider2* y se puede utilizar el método *query* para obtener un *Cursor*.
- A partir de las clases de *ContentResolver* y *ContentProviderClient* utilizando el mismo método *query*.
- Desde una *Activity*, se puede utilizar el método *managedQuery*.

Se decidió no controlar puntos de accesos deprecados a los contenidos, por lo que llamadas al proveedor deprecado de contactos *ContactsProvider* no serán controladas por ASA.

Consultando el código fuente de Android [**22**], se puede ver que entre las opciones enumeradas para acceder a contactos, tanto la que utiliza *CursorLoader* como *managedQuery*, terminan llamando a *ContentResolver*. Luego, tanto *ContentResolver* como *ContentProviderClient* realizan una query a la clase de *ContactsProvider2*, por lo que se decidió que éste es el módulo que finalmente se va a modificar.

ContactsProvider2 es el mediador entre las aplicaciones que quieren acceder a los contactos y las bases de datos que contienen los datos reales. Como toda consulta realizada al proveedor se resuelve en una query ejecutada por la clase *SQLiteDatabase*, podría analizarse la posibilidad de realizar las modificaciones en este punto.

Sin embargo, se decidió modificar a nivel del proveedor en lugar de modificar a más bajo nivel (a nivel de *SQLiteDatabase*) dado que:

- Al modificar el comportamiento para las consultas de *SQLiteDatabase* se tiene un punto global para todas los contenidos consultados que se encuentren almacenados en formato de tabla. Esto lleva a que sea

necesario identificar que tipo de contenido es el que se está intentando acceder en el momento en que se realiza la consulta, como por ejemplo datos del teléfono, contactos, etc, lo cual puede afectar la performance de todo el sistema y no escala a diferentes tipos de contenidos. Uno tendría que modificar el comportamiento de la clase *SQLiteDatabase* por cada recurso nuevo que se quiera administrar.
- Al ser *SQLiteDatabase* un punto de acceso global, las consultas modificadas son aquellas consultas realizadas sobre todas las bases de datos del sistema. Se debe contemplar el caso que la base de datos consultada sea una base de datos internas de una aplicación.
- Trabajar con proveedores no sólo permite modificar cómo se llevan a cabo las consultas para un tipo específico de contenido sino que también concede a ASA de atributos de calidad como la escalabilidad hacia otros contenidos y permite un fácil mantenimiento de código pues cada contenido tiene un punto de acceso particular.

Se debe modificar el método *query* para que sólo incorporen aquellos contactos que están incluidos en los grupos permitidos por ASA. Analizamos dos formas distintas de lograr esto:

- Implementando un *Cursor* propio, que sea polimórfico a *Cursor*, el cual al momento de ser iterado, saltée aquellas filas del resultado de la consulta que no pertenezcan a los grupos a los que la aplicación sí tiene acceso.
- Modificando la consulta para incluir como condición en la selección que los contactos en el resultado sean miembros de algunos de los grupos especificados.

```
final Cursor query(Uri uri, String[] projection, String
    selection,String[] selectionArgs, String sortOrder)
```

FIGURA 22. Prototipo del método *query* de *ContentProvider*

Se decidió implementar la segunda opción, debido a que contamos con los parámetros para realizar la query y es bastante intuitivo modificar la selección de la misma para filtrar los contactos que no pertenecen a los grupos a los que una aplicación dada tiene acceso.

Finalmente, hay distintas formas de modificar la clausula *WHERE* (selección) de la query original, entre las cuales fueron las consideradas:

- Realizar una operación de Join entre los miembros de los grupos especificados por el usuario en la política a aplicar y la tabla sobre la que se está realizando la consulta.
- Otra sería realizar una subquery dentro de la query original para consultar si el ID del miembro de esa fila está incluido en uno de los grupos permitidos.
- Finalmente, utilizar la operación IN para corroborar que la ID se encuentra en la lista de IDs de los miembros de los grupos permitidos.

Para simplificar la implementación, se eligió la última opción.

Primero sería necesario implementar una consulta que obtenga todos los miembros de los grupos permitidos por el usuario para la aplicación que accedió a los contactos. Un problema que se presentó es que no es posible realizar una consulta a la base de datos mediante el método query para otener dichos miembros, esto es porque este es el método que estamos hookeando por lo que se generaría una recursión infinita. Este es un problema típico de hooking que podría ser resuelto con un flag que indique que se trata de una versión modificada. Sin embargo, *Cydia Substrate* nos provee un método llamado *invoke* que nos permite realizar una llamada al método original y no al hookeado. De esta forma se pueden realizar las consultas necesarias sobre la API modificada a pesar de que se están realizando durante la ejecución de la llamada modificada, sin entrar en un ciclo infinito.

Uno debe tener particular cuidado sobre cómo modificar la consulta al proveedor original. La técnica elegida de agregar a la selección la condición de IN es de la forma

```
Columna IN (arreglo de identificadores de contactos
    permitidos)
```

Luego, necesitamos conocer cuál es el nombre de la columna que tiene el número identificador de contacto y la misma depende de la tabla sobre la cual se está realizando la consulta.

Las bases de datos consultadas utilizan distintos nombres de columnas para almacenar los identificadores de contactos. Existen tres bases de datos que guardan datos de contactos: *Contacts*, *Raw_contacts* y *Data*. Considerando que tenemos distintas tablas con distintas columnas debemos tener en cuenta que si accedemos a la tabla de *Contacts* se debe modificar la consulta sabiendo que la columna de interés para nosotros es *_ID*. Luego, modificaríamos la selección agregando la condición

```
_ID IN (arreglo de identificadores)
```

De la misma forma, consultas sobre *Raw_contacts* o *Data* usan la columna *CONTACT_ID*. Se puede detectar sobre qué tabla se está realizando la consulta utilizando el valor de URI que fue pasado como primer parámetro de la consulta.

4.2.4.3 El problema del contexto

En secciones anteriores se mencionó el problema de obtener un contexto para poder buscar la política asociada a una aplicación o identificar a la misma al momento en que intenta acceder al recurso cuyo acceso estamos modificando. En el caso del recurso de contactos esto no se presentó como un problema pues la clase modificada *ContactsProvider2* tiene un servicio público que nos devuelve el contexto sobre el cual el proveedor está corriendo. Luego, siempre podemos utilizar dicho contexto para identificar a la aplicación que

está corriendo utilizando los parámetros de UID y PID, y consultar a nuestro proveedor de configuraciones qué política se espera aplicar para dicha aplicación.

4.2.5. Datos del dispositivo

4.2.5.1 Granularidad

Luego de haber identificado algunos datos del teléfono que pueden potencialmente identificar tanto al dispositivo como a su dueño, según fue descripto en secciones anteriores, se tomó la decisión de que el control de dichos datos fuera lo más simple posible. Dado que tratamos con datos cuyas representaciones son atómicas, por ejemplo no tiene sentido un subconjunto de un IMEI, el intentar seguir el enfoque que se aplicó en contactos de definir su granularidad como un subconjunto de los datos reales no es algo posible. Es por eso que para cada valor atómico que represente alguno de los datos del dispositivo a controlar, se definió una forma particular de devolver un valor falso, uno aleatorio o especificado por el usuario, según éste haya decidido, de forma tal que sean polimórficos con los datos reales.

4.2.5.2 Implementación de control de acceso a información del dispositivo

Como se explicó previamente, la API de Android permite acceder a los datos del teléfono a través de la API de *TelephonyManager*. Cada uno de los métodos modificados devuelve una cadena de caracteres y, el resultado finalmente devuelto dependerá de la política establecida por el usuario.

Sin embargo, hay aplicaciones que pueden acceder a los datos del teléfono a través de clases internas de Android. Aunque son algunas pocas las aplicaciones que lo hacen, el acceso a las mismas constituiría en una violación a la política definida por el usuario si ASA no controlara su acceso. Investigando el código fuente de Android [**22**], vemos que la clase *Phone* permite obtener los mismos datos que la API de *TelephonyManager*. Más propiamente dicho, *Phone* es una clase abstracta donde sus dos subclases principales, *GSMPhone* y *CDMAPhone*, implementan los servicios que nos permiten obtener información del teléfono cuyo acceso pretendemos controlar con ASA, como ser el IMEI, IMSI y número de teléfono. Además, cada una de estas clases tiene más de una forma de obtener un mismo dato, como por ejemplo, el IMEI se puede obtener consultando al método *getDeviceID* o a *getIMEI*.

Como *TelephonyManager* utiliza la clase interna de *Phone* para consultar la IMEI, IMSI, número de linea o de SIM, no es necesario modificar los métodos de más alto nivel si modificamos el acceso a la información proveniente de la API interna.

Recordemos que fue una decisión de diseño importante en ASA el controlar el acceso a los recursos tanto para aplicaciones de terceros como del sistema. Luego de confirmar que el sistema usa la API de *Phone*, ASA hookea los métodos correspondientes de esta última, en vez de hookear a nivel de *TelephonyManager*.

CUADRO 1. Mapa de métodos que obtienen la misma información del dispositivo

Información	Phone	TelephonyManager
IMEI	getDeviceID getIMEI (GSM) y getMEID(CDMA)	getDeviceID,
IMSI	getSubscriberID	getSubscriberID
Número de SIM	getIccSerialNumber	getSIMSerialNumber
Número de teléfono	getLine1Number	getLine1Number

En la tabla 1 se presentan cuáles fueron los métodos de la clase *Phone* modificados por ASA para controlar el acceso a los datos del teléfono. Además, agregamos una columna con meros motivos informativos que incluye los métodos equivalentes de más alto nivel en *TelephonyManager* que, como ya dijimos, no fueron modificados por ASA.

En cada uno de los métodos modificados se identifica a la aplicación que está queriendo acceder al recurso del teléfono utilizando su identificador de usuario (UID) y su identificador de proceso (PID). Esto se realiza antes de obtener el valor real que devuelve el servicio original que estamos modificando. Luego, se realiza una consulta al proveedor de preferencias de ASA de forma de obtener la política del usuario a adoptar para la aplicación en cuestión. En caso de no haber una configuración particular para la misma, se obtiene la configuración por defecto correspondiente.

La configuración para cada aplicación se proyecta en una columna de la base de datos de configuraciones por cada propiedad modificada. Por ejemplo, si el usuario decide que quiere cambiar su *IMEI* para que una aplicación dada vea un valor aleatorio (es decir, un valor falso cada vez que la aplicación realice una consulta a dicho valor), entonces aparecerá el valor *Random* (del inglés, Aleatorio) en la columna *getDeviceId* en la fila correspondiente a dicha aplicación y en la tabla del recurso de datos del teléfono, que nosotros llamamos *device_data_settings*. Si el usuario especifica un IMEI manualmente, dicho valor será almacenado en su registro correspondiente. En los casos en los que hay varios métodos que nos proveen el mismo dato, por ejemplo del identificador del dispositivo *IMEI*, es necesario realizar un mapeo para

obtener el nombre de la columna correspondiente.

Esta decisión de diseño facilita el acceso a la configuración, dado que podemos usar el nombre del método modificado en la selección de la consulta a la base de datos de ASA de forma tal de obtener sólo el valor de interés y no toda la fila de la tabla sobre la que hacemos la consulta. Los posibles valores en estos campos son: Real, Fake o Random y, para cada una de estas configuraciones, se devuelve a la aplicación un valor generado particularmente para cada caso.

Hay tres servicios de la API de *TelephonyManager* que fueron modificados, dos de los cuales nos permiten obtener información de la torre de celular a la que el dispositivo se encuentra conectado o de las torres vecinas. Sin embargo, debido a que dichos servicios son utilizados para obtener información de geolocalización del dispositivo y son administrados por ASA para controlar el acceso del resto de las aplicaciones a dicha información, se dará más detalle de los mismos cuando se explique la implementación del módulo de *geolocalización* 4.2.7.

4.2.5.3 Aleatorio vs Falso

Entrando un poco en detalles más implementativos, vemos interesante destacar cómo son calculados los valores aleatorios o falsos para cada uno de los valores modificados. No está de más aclarar que cuando la configuración es marcada como real, ASA se limitará a simplemente devolver el valor real consultado. Siguiendo el mismo enfoque, en el caso en que el usuario haya definido un valor *Custom*, se usará dicho valor como resultado de la consulta.

La opción de definir un valor como falso se implementó usando una función llamada *fakeNumberFrom* que nos permite obtener un valor que aparente ser real a partir del valor verdadero del dispositivo.

Usamos el valor real para determinar el falso utilizando SHA256. Éste es un algoritmo de hashing suficientemente robusto para no permitir obtener el valor real a partir del falso. Además, es inútil utilizar tablas precomputadas para intentar obtener el valor real ya que por un lado sólo se utiliza un prefijo de la función de hash, cuya longitud coincide con la del valor original. Por otro lado, la implementación de *fakeNumberFrom* toma el módulo diez de cada byte generado por la función de hash por lo cual un entero del valor falso puede provenir de al menos 25 bytes diferentes. Es importante notar que el resultado es siempre igual, para un mismo dispositivo, dado que se utiliza como semilla el valor original. Esta propiedad podría ser útil si un usuario quiere ser identificado por dos aplicaciones con el mismo *IMEI* y que las mismas sepan que se trata del mismo dispositivo pero, a su vez, no dar el valor real del mismo. Esta propiedad no es posible si definimos este valor como aleatorio.

```
private static String fakeNumberFrom(String seed) {
        MessageDigest digest= null;
```

```
        try {
                digest = MessageDigest.getInstance("SHA-256");
                digest.update(seed.getBytes("UTF-8"));
        } catch (Exception e) {
                Log.d(Utilities.ERROR, "An error occurred
                    generating a fake number");
                }
        byte[] hash = digest.digest();

        char[] fakeChars = new char[seed.length()];
        for (int i = 0; i < fakeChars.length; i++) {
                int n = Math.abs(hash[i]);
                fakeChars[i] =  Character.forDigit(n % 10, 10);
        }

        return new String(fakeChars);
}
```

Por último, la opción de definir un valor como aleatorio se implementó generando un valor pseudo-aleatorio teniendo en cuenta la longitud original del valor real. Por ejemplo, para el número de SIM se utilizará un generador aleatorio de 19 dígitos, dado que el valor original del número serial de una tarjeta SIM tiene ese formato.

Debido a que más de uno de los métodos modificados obtienen un número aleatorio de longitud n, implementamos un método llamado *randomNumberWithLength* que nos provee este número en forma de cadena de caracteres, de forma tal de ser consistentes con el valor que espera obtener la aplicación en cuestión.

Notar que este método llama al descripto anteriormente *fakeNumberFrom*. Esto se debe a que un número de tipo Long de Java tiene un máximo de 19 dígitos y, como queremos un número de longitud arbitraria, si n es mayor a 19 obtendríamos un número con muchos ceros adelante. Para evitar esto, creamos primero el número random y luego dejamos que *fakeNumberFrom* termine el trabajo de hacer que el número no tenga dígitos cuyo valor se pueda preveer.

```
private static String randomNumberWithLength(int n) {
        Random rand = new Random();
        Long r = Math.abs(rand.nextLong());
        String result = String.format("%0" + n + "d", r);

        return fakeNumberFrom(result.substring(0, n));
}
```

Para ambos valores, tanto el aleatorio como el falso, suponemos que la longitud del valor que estamos hookeando tiene un máximo de 32 dígitos. Esto se debe a que, como usamos SHA256, tenemos sólo 32 bytes de los cuales obtenemos un dígito por cada uno. En el caso de los valores hookeados, el que tiene mayor cantidad de dígitos es el número serial de la tarjeta SIM

que tiene 20 dígitos.

4.2.5.4 El problema del contexto

Como explicamos anteriormente, muchas veces logramos obtener un contexto válido presuponiendo que la aplicación que quiere acceder al recurso controlado tuvo que haber hecho una llamada a alguno de los servicios de la API de *Context*. En el caso del recurso de datos del dispositivo, podemos asegurar que, para obtener la instancia actual de *TelephonyManager*, una aplicación tuvo que haber utilizado el método *getSystemService* de la clase *Context*. Modificando este método, podemos quedarnos con un contexto váalido de forma transparente para la aplicación.
Cuando se modifica la clase *Phone*, el contexto de la aplicación se obtiene a partir de su método público *getContext* (utilizando *Java Reflection* debido a que se trata de una clase interna de Android). De esta forma nos aseguramos tener siempre un contexto no nulo y válido.

4.2.6. Redes Inalámbricas

4.2.6.1 Granularidad

La implementación del módulo de acceso a información de redes inalámbricas tiene algunas similitudes con la del módulo anterior. Muchos de los valores que ASA controla son atómicos (como el SSID, BSSID, etc) por lo que no es posible obtener un subconjunto de valores como se hacía en el módulo de contactos. Es por eso que, para estos valores atómicos, la política que puede definir el usuario está limitada a dar un valor real o uno falso, con la posibilidad de modificar este último a mano, siempre y cuando cumpla con la especificación del formato esperado para dicho valor.

Sin embargo, hay dos propiedades administradas por nuestro sistema que no son atómicas. Estas propiedades son: la lista de todas las redes previamente configuradas en el dispositivo y la lista de redes inalámbricas escaneadas en el último scan del punto de acceso (AP) al que el dispositivo está conectado.

4.2.6.2 Guardando una configuración de WiFi

Una funcionalidad importante al momento de engañar a una aplicación que accede a los datos provenientes del WiFi del equipo es la de permitir al usuario guardar una configuración de WiFi particular para luego utilizarla al momento de definir la política de una aplicación. Por ejemplo, uno podría guardarse la configuración de la red inalámbrica que usa en su oficina y luego usarla desde su casa para que una aplicación obtenga estos datos en vez de los reales. Además, el usuario tiene la posibilidad de editar los valores de una configuración guardada al momento de definir la política a aplicar

para una aplicación. De esta forma se le da mayor poder al usuario que el obtenido definiendo una configuración con valores falsos.

Para guardar la configuración de una red WiFi, ASA primero se asegura de que el dispositivo se encuentre conectado a alguna red inalámbrica. Después, se guarda en una tabla de estados de WiFi los valores que se obtienen al consultar cada uno de los datos controlados por ASA. Se guarda una copia de los valores atómicos como SSID, BSSID, etcétera, mientras que para las redes escaneadas y las configuradas se almacenan los SSIDs de cada una de las configuraciones de las respectivas listas de configuraciones inalámbricas. Si un usuario luego define una política en base a una configuración guardada, los valores en los campos de la tabla de estados de WiFi se copian a la tabla donde se definen las políticas para el recurso de redes inalámbricas agregando el campo donde se define para qué aplicación se está definiendo.

4.2.6.3 Implementación de control de acceso a información de redes inalámbricas

Todas las propiedades atómicas, es decir, aquellas indivisibles, como ser la dirección MAC del equipo, se obtienen a través de una instancia de la clase *WifiInfo*, que se obtiene al llamar al método *getConnectionInfo* del proveedor *WifiManager* y que es, básicamente, una estructura que contiene toda la información relacionada con una configuración de una red inalámbrica particular. Es por eso que modificamos la API de *WifiInfo* para controlar el acceso a esta información.

Las propiedades nombradas que no son atómicas sino una lista de configuraciones de redes inalámbricas, son servicios provistos por la API de *WifiManager*, por lo que también se modificó la misma para completar el control de ASA sobre la información proveniente de la configuración de redes inalámbricas del equipo.

Además de estos métodos, se puede obtener la dirección IP y dirección MAC del equipo utilizando la clase *NetworkInterface* de Java. Por último, como se mencionó en los conceptos teóricos de WiFi, hay otras dos formas de obtener la dirección IP del dispositivo utilizando un servicio provisto por la clase privada *Utils* de Android,*getDefaultIpAddresses* y *getWifiIpAddresses*. Estos métodos son usados por el sistema para obtener dicha propiedad cada vez que ingresamos a *Settings* $\Rightarrow$ *WiFi* $\Rightarrow$ *Advanced Settings*.

Un pequeño detalle implementativo es que no todos los servicios que devuelven la dirección IP del dispositivo son polimórficos. Algunos devuelven una cadena de caracteres formateada como una dirección IP (4 grupos de un byte cada uno, separados por un punto) y otros devuelven un número entero que es el equivalente a esos mismos bytes. Es por eso que se decidió que nuestra base de datos trabajase con la cadena de caracteres y se implementó un servicio llamado *getRealIPFrom* que toma una cadena con

el formato mencionado y sabe cómo obtener el valor numérico asociado a dicha cadena:

```
private static int getRealIPFrom(String ipAddress) {
        String[] octets = ipAddress.split("\\.");
        return (Integer.parseInt(octets[3]) << 24) + (Integer.
            parseInt(octets[2]) << 16) + (Integer.parseInt(octets
            [1]) << 8) + Integer.parseInt(octets[0]);
}
```

Por el momento, cuando un usuario elige valores falsos, los mismos se encuentran predefinidos por ASA. Así, por ejemplo, el SSID falso utilizado por ASA es *WiFi* mientras que el BSSID es *bb:bb:bb:bb:bb:bb*. Recordemos que, si un usuario quiere poder ingresar valores distintos a los definidos por este modo de ASA, debería hacer uso de las ventajas de guardarse una configuración y modificarla según desee.

En el caso en que un usuario defina una política para una aplicación usando una de las configuraciones guardadas en ASA, para el caso en que se intente acceder a uno de los datos atómicos, ASA simplemente devuelve el valor que se encuentra en el campo que corresponde al método consultado para la aplicación que está realizando la consulta. En caso de que se trate de las redes configuradas, ASA filtra las redes configuradas almacenadas del dispositivo de modo tal que sólo queden aquellas que aparecen en la lista de SSIDs guardadas en la configuración que se está aplicando. Por otro lado, con las redes escaneadas, cuando un usuario guarda una red, además de guardarse la información de la red inalámbrica actual, se almacena la información de las redes WiFi escaneadas en ese momento. Es decir, se guarda para cada red WiFi escaneada todos los datos de una instancia de *ScanResults*. Luego, a partir de estos valores, ASA puede reconstruir las redes escanedas para que sean devueltas a una aplicación que tenga definido una política utilizando una red guardada. Esta funcionalidad permite mentirle a una aplicación acerca de la ubicación del dispositivo, cuando la misma utiliza las redes escaneadas para obtener tal fin.

4.2.6.4 El problema del contexto

Así como en el caso del recurso de datos del dispositivo, logramos obtener un contexto válido presuponiendo que la aplicación que quiere acceder al recurso controlado tuvo que haber hecho una llamada a alguno de los servicios de la API de *Context*. En el caso del recurso de redes inalámbricas hallamos que para obtener la instancia actual de *WifiManager*, una aplicación tuvo que haber utilizado el método *getSystemService* de la clase *Context*, de forma similar a como en el caso del recurso de datos del dispositivo obtenía una instancia de *TelephonyManager*. Aprovechando que ya tenemos este método modificado, nos guardamos la instancia actual del contexto de la aplicación, también para el recurso descripto en este último módulo.

4.2.7. Geolocalización

4.2.7.1 Granularidad

Dado que este módulo intenta controlar el recurso que provee información de la ubicación del equipo, y como una ubicación está representada por un conjunto de valores atómicos y es atómica en sí, todos los datos de geolocalización controlados por ASA son guardados, dado que no existe el concepto de subconjunto para un punto geográfico. Sin embargo, para seguir con el espíritu de ASA de dar poder al usuario para definir *qué parte de toda la información real* una aplicación puede acceder (recordemos que queremos definir un subconjunto de los datos reales), decidimos agregar una variable más que controla la precisión de la ubicación definida en una política. Por ejemplo, un usuario podría decidir indicar su ubicación a la aplicación *Weather Channel* que dada una ubicación nos indica el estado del clima, como su ubicación real pero con una precisión mala como para recibir información del estado del tiempo en el lugar donde se encuentra pero sin necesidad de indicar su posición geográfica exacta.

4.2.7.2 Definiendo una ubicación falsa

ASA provee al usuario dos formas de definir una ubicación falsa. Una de ellas usa la API de *Google Maps* de forma tal de mostrar un mapa al usuario donde este pueda elegir un punto en el mismo.

Otra opción, de forma similar a la funcionalidad de redes inalámbricas, es la de poder guardar toda la información que define una ubicación en un momento dado para el dispositivo. Luego, el usuario podría usar esta ubicación guardada a la que llamamos estado de ubicación, para definir la política de geolocalización de alguna aplicación. Por ejemplo, podríamos guardarnos la ubicación del bar de la esquina de casa, irnos de vacaciones a China y mentirle a *Facebook* para que crea que nunca dejamos el bar.

Para guardar la configuración de una ubicación, ASA intenta obtener la ubicación actual por todos los medios que tiene según las propiedades de hardware y software del dispositivo. Luego, guarda esta información en una tabla de estados de ubicación de forma tal de poder reconstruir la ubicación guardada con esos datos. Notar que usando esta funcionalidad, ASA tiene poder para guardar un estado de ubicación completo que incluye la posición actual según el servicio de GPS (latitud y longitud), la lista de redes inalámbricas escaneadas y la torre de celulares según el área que corresponda. Con esto nos referimos a que si el usuario quiere definir una ubicación falsa marcando un punto en el mapa, ASA no tiene forma de obtener información real de las torres de celulares o las redes inalámbricas escaneadas, por lo que estos valores son definidos como no existentes de forma tal de forzar a que una aplicación que quiere acceder al recurso use la ubicación definida por el servicio de GPS. Una implementación a futuro, es contar con una base de datos colaborativa de estados pero ASA no cuenta con esta

funcionalidad aún.

4.2.7.3 Implementación de control de acceso a la geolocalización del dispositivo

Se utilizó un celular inteligente donde probar nuestro sistema cuya versión de Android es *4.0.4* . El dispositivo que veníamos usando para contactos, no pudo ser utilizado para realizar pruebas en el recurso de geolocalización, dado que la tablet no tenía hardware de GPS.

Existen distintas fuentes que definen la ubicación del dispositivo y dichos valores deben ser coherentes entre sí de modo tal de representar la misma ubicación final. Es por eso que, para una política definida por el usuario para una aplicación que tiene acceso al recurso de geolocalización, se deben definir los elementos utilizados por cada proveedor de este recurso para determinar una ubicación, de forma tal que cada uno de ellos resuelva la posición geográfica del dispositivo de forma no contradictoria con el resto.

Además, esta coherencia debe ser mantenida por el modelo de ASA que se encarga de, para cualquier forma en la que el usuario haya definido una ubicación falsa (ya sea marcando una ubicación en el mapa o usando una ubicación guardada), devolver como resultado de cualquier consulta de una misma aplicación, ubicaciones coherentes entre sí.

Según secciones anteriores donde describimos las formas en las que una aplicación puede obtener la ubicación del dispositivo, resaltamos dos métodos principales, independientemente de la fuente de información. Es por eso que ASA debe atacar ambos flancos para asegurar un funcionamiento análogo y en todo momento, sin importar las capacidades del hardware o software del dispositivo.

En el caso del método *getLastKnownLocation*, una vez identificada la aplicación, se devuelve una ubicación falsa reconstruida a partir de la latitud y la longitud de la política definida por el usuario (recordemos que el usuario puede decidir devolver la ubicación real del dispositivo, y en este caso, ASA no modifica el valor original de la consulta).

Para el segundo método donde una aplicación puede anotarse para recibir una o más actualizaciones de la ubicación del dispositivo usando los servicios *requestSingleUpdate* y *requestUpdates*, respectivamente, implementamos una técnica que es conocida como *man in the middle* (hombre en el medio, en español) donde básicamente intervenimos entre el objeto que originalmente se anotó a recibir dichas actualizaciones y el objeto que genera las mismas. Para esto, implementamos un *LocationListener* falso cuya tarea es intervenir cada vez que se reciba una actualización en la ubicación, notificando al *LocationListener* original usando la ubicación definida según la política. De esta forma podemos controlar de forma rigurosa qué información llega a cada aplicación cada vez que hay un cambio en la ubicación

geográfica del dispositivo.

Como ya sabemos, las redes inalámbricas escaneadas del dispositivo son utilizadas para obtener una ubicación. Por eso, es necesario que el usuario de ASA, al momento de definir una política de geolocalización, sea consciente que debe modificar el acceso a la información de WiFi, de forma tal de ocultar estas redes a la aplicación que se quiere controlar. Se decidió que ASA no oculte automáticamente esta información, para no crear dependencias o inconsistencias entre los distintos recursos.

Por último, otra fuente para controlar los accesos a la ubicación del dispositivo es la de las torres de celulares. Esta información es manejada por el gestor *TelephonyManager* a través de tres servicios distintos que exponen estos datos: *getCellLocation*, *getNeighboringCellInfo* y *listen*.

En el caso de *getCellLocation*, ASA reconstruye una instancia de *CellLocation* a partir de los datos almacenados en caso de que el usuario haya definido una política a partir de un estado de ubicación. Estos datos dependen del tipo de torre al que el dispositivo estaba conectado en momento en que la ubicación fue guardada. Por ejemplo, puede tratarse de una torre GSM, CDMA o nula, siendo esta última para el caso de que el dispositivo no pueda conectarse a una torre. A pesar de que la API de Android no utiliza la información de *getNeighboringCellInfo* para determinar la ubicación geográfica a partir de torres de celulares vecinas, ASA devuelve un conjunto vacío de torres en caso de que se haya definido una política diferente a la de obtener la ubicación real. El método *listen* nos provee de un servicio general para escuchar diferentes eventos relacionados con información del equipo. Es por eso que ASA modifica el acceso cuando una aplicación se registra al evento que escucha el estado de la torre de celulares a la que está conectado el dispositivo. Se procede de forma similar a *requestUpdates* implementando un *PhoneStateListener* falso que controla dicho acceso encargándose de informar de cambios en la ubicación en el dispositivo con los datos definidos en la política que corresponde a la aplicación.

4.2.7.4 El problema del contexto

Así como en el caso de los recursos de datos del dispositivo y redes inalámbricas, logramos obtener la instancia actual de *LocationManager* reutilizando el método *getSystemService* modificado.

4.2.8. Proveedor de contenido y base de datos de configuración de aplicaciones

Nuestro sistema tiene una base de datos por cada recurso administrado por ASA donde guardamos las configuraciones, previamente especificadas por el usuario, de las aplicaciones que hacen uso de dichos recursos.

Dado que las bases de datos del sistema implementado se encuentran en el espacio local de ASA, no fue necesario implementar en nuestro proveedor de contenido propio servicios para agregar, actualizar o borrar registros de la base al momento en que el usuario modifica alguna de las políticas de control usando nuestra interfaz de usuario. Estos servicios son provisto por la API de SQLite de Android. Sólo se implementó un servicio propio para poder realizar una consulta desde el espacio local de las aplicaciones modificadas.

4.2.8.1 Nuestro proveedor de contenido: ASASettings

Sabemos que la extensión de código a la API de Android debe poder consultar las políticas definidas por el usuario que se encuentran en la base de datos de ASA. Dado que el hook corre en el proceso de la aplicación y que estas políticas no se encuentran en el espacio local de la misma, se implementó un proveedor de contenido propio que denominamos *ASASettings*. La idea de este proveedor es permitir que nuestro hook pueda acceder a las bases de datos de ASA a través de una comunicación interproceso.

Hay dos observaciones claves para hacer respecto a este proveedor: Como se mencionó en la sección de *Conceptos Previos*, un proveedor de contenido puede definir sus propios permisos para restringir el acceso a sus datos, de forma de poder denegarle el acceso a aplicaciones que no tengan dicho permiso. Vale aclarar que esto no puede ser implementado para el proveedor de ASA, *ASASettings* debido a que nuestra modificación a la API de Android corre en el espacio de una aplicación que no definió ese permiso en su manifiesto y ASA no define esos nuevos permisos para las aplicaciones a controlar. Es por eso que se decidió que cualquier aplicación puede realizar consultas a *ASASettings*. *ASASettings* sólo implementa el servicio *query* para que una aplicación externa sólo pueda realizar consultas a la base de datos de ASA y no inserciones, borrados o modificaciones a la misma. De esta forma, evitamos que una posible aplicación maliciosa pueda modificar su propia política de ASA desde su espacio cambiando la base de datos a través de nuestro proveedor.

Por ejemplo, el hook de *TelephonyManager* corre en un proceso determinado. Al momento de necesitar la política definida para los datos del teléfono, no puede acceder a ellas a través de una query a *SQLite* porque la base de datos no se encuentra en el espacio local del proveedor. A su vez si hubiésemos agregado un permiso para restringir el acceso a *ASASettings*, el proveedor *TelephonyManager* tampoco podría obtener la política mediante IPC debido a que no tiene declarado este permiso en su manifiesto.

El hecho de que cualquier aplicación pueda realizar consultas a nuestro proveedor, tiene como desventaja que una aplicación consciente de que ASA se encuentra instalada puede consultar por cualquier política definida en la base de datos de *ASASettings*.

A fin de restringir las consultas de una aplicación a sólo sus políticas se probaron dos posibles enfoques. El primero consiste en modificar *ASASettings* para que sólo acepte consultas de los procesos de los proveedores de los recursos controlados por ASA (*ContactsProvider*, *WiFiManager*, *LocationManager*, etc). El segundo enfoque consiste en que nuestro proveedor verifique que el UID y el PID de la política consultada coincidan con el UID y el PID de la aplicación que realizó la consulta. Ninguno de estos dos enfoques descriptos cumplen con su objetivo debido a que no todos los métodos hookeados corren en procesos diferentes al de la aplicación que realizó la llamada. Por ejemplo, en el caso del proveedor de contactos, el método *query* corre en un proceso distinto al de la aplicación que quiere acceder a los contactos, en cambio, los métodos de la clase *WifiInfo* corren en el mismo proceso. Entonces, el segundo enfoque no funciona para el método *query* mientras que el primero no funciona para los métodos de *WifiInfo*.

4.2.8.2 Estructura de la base de datos

Para almacenar nuestra propia base de datos utilizamos la clase *SQLiteOpenHelper*, una clase de Android de ayuda que nos permite una fácil creación y manejo de base de datos. Dicha clase creará un archivo *settings.db* almacenado en el directorio *databases* dentro del espacio de nuestra aplicación. Dicho archivo contiene una tabla para cada recurso y una adicional para las configuraciones por defecto.

Cada tipo de recurso cuenta con una base de datos donde se almacena la configuración especificada para cada aplicación que lo acceda y cuya política particular fue definida por el usuario. Si bien cada tabla es específica a cada tipo de recurso, todas siguen un patrón común que nos permite identificar unívocamente a una aplicación. Para esto, todas las tablas de recursos tienen dos columnas principales para identificar a una aplicación, *pkgName* y *processes*. En la sección en donde se describe cómo identificamos una aplicación 4.1.2 se explicó bien el por qué de dichas columnas y el significado de la información que contienen.

Sin embargo, dado que no necesitamos configurar aplicaciones particulares en la tabla de configuraciones por defecto, este patrón no está en esa tabla.

En el caso de los de contactos, la configuración se guarda en una sola columna donde se serializa la lista de grupos de contactos a los que la aplicación en cuestión tiene acceso para luego deserializarla al momento de tener que aplicar esta configuración. Por ejemplo, si tenemos tres grupos *Amigos*, *Familia* y *Trabajo* con identificadores 1, 2 y 3 respectivamente, la configuración de una aplicación que sólo puede acceder a los grupos de *Amigos* y *Trabajo* en la base de datos sería la serialización de la lista compuesta por los números 1 y 3.

En cambio, para recursos como geolocalización, datos del dispositivo e información de redes inalámbricas, que tienen múltiples ejes de configuración, se agrega una columna que guarde la configuración para cada uno de estos ejes.

4.2.8.3 Tablas adicionales

ASA tiene algunas tablas adicionales donde se guarda información proveniente de configuraciones que denominamos *custom* (*a medida*, en español) para el recurso de información de redes inalámbricas y el de geolocalización. Para el primer recurso, la tabla adicional guarda lo que llamamos estados de una configuración WiFi en un momento dado. De esta forma, este estado podría ser usado en la configuración de la política de alguna aplicación.

Respecto al recurso de geolocalización, la tabla adicional guarda toda la información respectiva a la ubicación del dispositivo en un momento dado. También llamamos estados al conjunto de datos de una ubicación que representa una configuración *custom* que el usuario guardó en un momento dado. Al igual que el caso de WiFi, estos estados pueden ser usados para definir la política de geolocalización de una aplicación.

Cuando un usuario pide almacenar la información de la red inalámbrica o su ubicación actual, se crea una nueva entrada en la base de datos correspondiente con los datos necesarios para que la configuración pueda ser reconstruida a partir de los mismos en el momento en que tenga que ser usada para engañar a una aplicación. Por ejemplo, es necesario almacenar el SSID, BSSID, dirección de IP, MAC, redes escaneadas y configuradas para poder definir una política válida del recurso de WiFi. Si bien la MAC no cambia, se almacena la misma pues se permite al usuario editar cada campo de una configuración guardada de modo tal de permitirle definir políticas más flexibles.

La funcionalidad de almacenar una ubicación válida del dispositivo es clave para poder manipular al proveedor *NetworkProvider*, pues es la única forma que tiene actualmente el sistema de obtener una instancia de *CellLocation* válida que represente la información de una torre de celulares.

4.2.8.4 Tablas de configuraciones por defecto

El sistema tiene una base de datos donde guarda las configuraciones por defecto a aplicar cuando no se encuentra una particular para una aplicación. A dicha tabla se le dio el nombre de *default_settings* y contiene un campo que indica si se trata de la configuración por defecto para aplicaciones del sistema o para aplicaciones de terceros.

Como se guardan las preferencias por defecto para todos los recursos administrados por ASA se debe contar con un esquema simple para que

se pueda almacenar cualquier posible configuración para todos los recursos controlados. Es por eso que el esquema de configuraciones por defecto contiene una columna para identificar al recurso (se almacena el nombre de el o los permisos relacionados con dicho recurso), otra que indica si se trata de las políticas por defecto de las aplicaciones de terceros o del sistema, y otra con la configuración almacenada en sí serializada de forma tal de poder deserializarla al momento en que debe utilizarse.

El contenido de la columna de configuración dependerá del recurso asociado a una fila particular de la tabla. Para aquellos recursos cuyas tablas de configuraciones contengan más de una columna, la configuración por defecto será una lista serializada que contiene un valor por cada columna de la tabla del recurso en cuestión separando cada valor con una coma. En el caso de que uno de los valores sea una lista, esta es serializada de forma tal de que los valores de la misma estén separados por una coma y toda la lista encerrada entre corchetes.

4.2.9. La dificultad de controlar el acceso de aplicaciones del sistema

En Android, hay aplicaciones que tienen acceso a clases internas del firmware no provistas por la API de Android. No sólo aplicaciones del sistema pueden acceder a las mismas, aunque no es trivial para aplicaciones de terceros hacerlo [**1**]. Como el objetivo del sistema propuesto es poder controlar el acceso de todo tipo de aplicaciones, incluyendo las del sistema, es necesario tomar los recaudos necesarios. Un ejemplo de estas aplicaciones que tienen acceso a estas clases internas es la aplicación de ajustes del equipo conocida comúnmente como *Settings*. Esta aplicación del sistema permite al usuario visualizar y modificar las configuraciones del dispositivo. Como se mostrará más adelante en el capítulo de experimentaciones realizadas, *Settings* utiliza la clase interna *Phone* para obtener el identificador del dispositivo (IMEI o MEID, según corresponda). Si ASA no controlara el acceso a estas clases internas, la información privada del usuario podría ser obtenida a través de estos canales.

Como es de esperarse, la información que se puede obtener en algunas de estas clases privadas es la misma que la que se puede obtener a través de los proveedores de recursos que encontramos en la API pública de Android. Incluso, muchas veces, los proveedores de recursos son sólo *Wrappers* (del inglés, envoltorios) de ellas. Esto se mencionó previamente en la sección del recurso de información del dispositivo, donde tanto en la clase *Phone* como en *TelephonyManager* se podían obtener el IMEI, IMSI o número de línea.

En esta situación, se tomó la decisión de controlar el acceso modificando los métodos de las clases internas y no los de aquellas clases de la API que las envuelven, por cuestiones de eficiencia.

Otro caso que puede ser tomado como ejemplo es el de la clase interna *Utils* que permite obtener la dirección IP del dispositivo mediante los

métodos *getWifiIPAddresses* y *getDefaultIPAddresses*, ambos utilizados por la aplicación del sistema *Settings* para obtener esta información. Sin embargo, en este caso, el método para obtener la IP utilizando el proveedor *WifiManager* no utilizan los métodos internos de *Utils* por lo cual ambas vías de información deben ser controladas por ASA de forma simultánea.

Capítulo 5

Experimentaciones

En esta sección se realizaron pruebas para cada uno de los recursos controlados por ASA. Para esto se utilizaron aplicaciones que usan dichos recursos y se analizaron distintos aspectos que incluyen tanto entender a la aplicación analizada en tiempo de ejecución como al estudio de su código fuente, entre otros. En algunos casos, se utilizaron aplicaciones que nos permiten realizar las pruebas pertinentes.

Todas las pruebas se realizaron sobre dos dispositivos, un Sony Xperia S con Android 4.0.4 y un Samsung Galaxy SIII con Android 4.1.2.

5.1. Agenda de contactos

5.1.1. WhatsApp

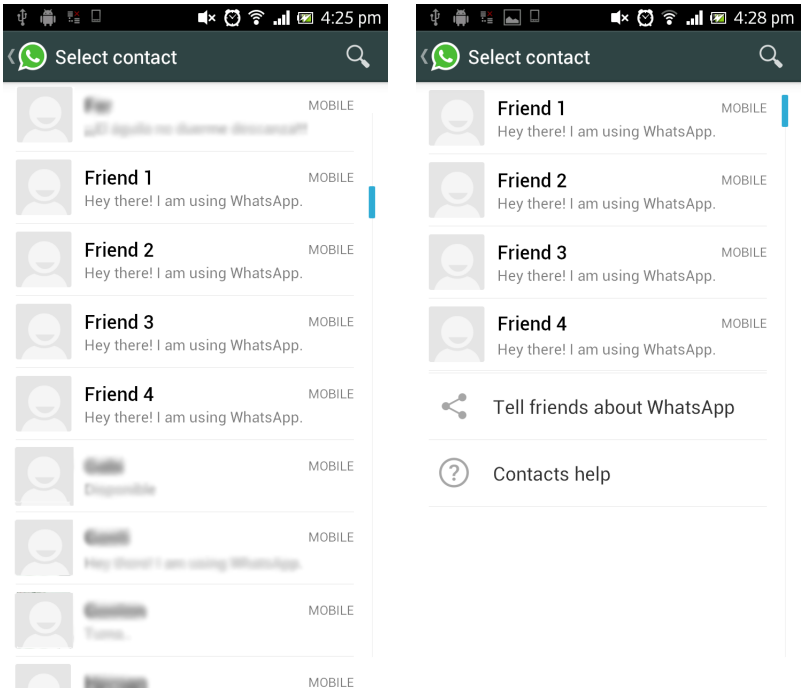
WhatsApp Messenger es un mensajero para teléfonos inteligentes disponible para Android y otros sistemas operativos. El mismo usa servicio de 3G o WiFi (de estar disponible) para enviar mensajes a contactos. Además de enviar y recibir mensajes de texto, se pueden enviar imágenes, audios y videos.

Utilizamos ASA para controlar el acceso a nuestra agenda de contactos para esta aplicación. En una primera instancia se dejó que acceda a todos los contactos en la agenda y luego limitamos el acceso sólo a los contactos que se encuentren en el grupo de *Amigos*, como puede verse en la figura 23. Como se explicó en secciones anteriores, éste es un claro caso en el cual ASA puede limitar el acceso de una aplicación a un subconjunto del recurso original de forma tal de balancear qué tanta información le damos a la aplicación que quiere acceder al recurso y el poder seguir utilizando la aplicación para los fines para los cuales la instalamos en nuestro equipo en un primer momento.

5.1.2. Por qué es necesario dar soporte para otras versiones

La idea de esta sección es mostrar que ASA está preparada para controlar aplicaciones en Android 4.0.3 y 4.0.4. Sin embargo, no podemos asegurar un correcto comportamiento en otras versiones de la API de Android, dado que la misma va cambiando con el tiempo y muchas de las aplicaciones pueden estar usando una versión distinta de la misma.

Para esto utilizamos como prueba a la aplicación *WhatsApp*. Luego de inspeccionar el código fuente de la API de Android para la versión 4.0.4 y



(a) Todos los contactos (b) Grupo de Amigos

FIGURA 23. Controlando el acceso a contactos de *WhatsApp* (Sony Xperia)

luego para Android 4.1.2, podemos ver que nuestro hook no funciona al momento de controlar aplicaciones que acceden al recurso de contactos porque la API cambió (siempre y cuando la aplicación en cuestión use esta nueva versión, dado que Android provee compatibilidad hacia atrás hasta cierta versión).

Finalmente, podemos afirmar que, para Android 4.1.2, no funciona nuestra modificación al método query de la clase *ContactsProvider2*.

Inspeccionando el código fuente [**22**] de la versión del método query para la versión 4.0.4 podemos ver que su prototipo es:

```
public Cursor query(Uri uri, String[] projection, String
    selection, String[] selectionArgs, String sortOrder)
```

Y el método query de la versión 4.1.2:

```
public Cursor query(Uri uri, String[] projection, String
    selection, String[] selectionArgs, String sortOrder,
    CancellationSignal cancellationSignal)
```

Además, la versión de query de Android 4.1.2 también tiene la versión de Android 4.0.4 para conservar compatibilidad hacia atrás. Lo único que cambia en la 4.1.2 es su implementación que llama a la nueva query con el parámetro adicional *cancellationSignal* nulo.

Supusimos que la razón por la cual el acceso de *WhatsApp* a contactos no es modificado correctamente por ASA es porque esta aplicación hace uso de la nueva versión de la API. Tenemos dos formas de comprobar este supuesto. La primera es desensamblar el apk de *WhatsApp* para Android 4.1.2, lo que tomaría mucho trabajo, dado que el código de *WhatsApp* está ofuscado. La segunda forma es hookeando esta nueva llamada en ASA y ver que *WhatsApp* efectivamente usa la nueva versión de query en lugar de la que ASA controla.

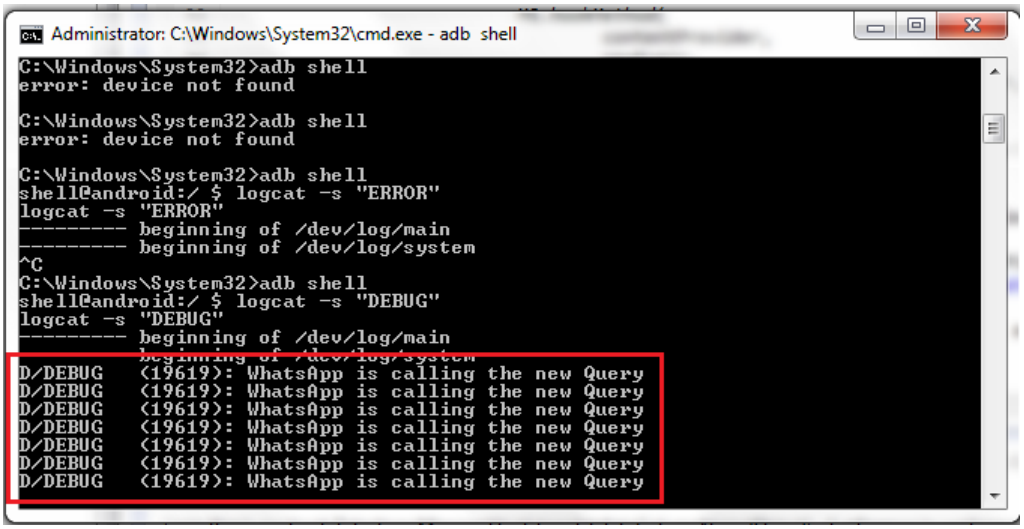


FIGURA 24. *WhatsApp* llama a la nueva versión de query para Android 4.1.2 (Samsung Galaxy SIII)

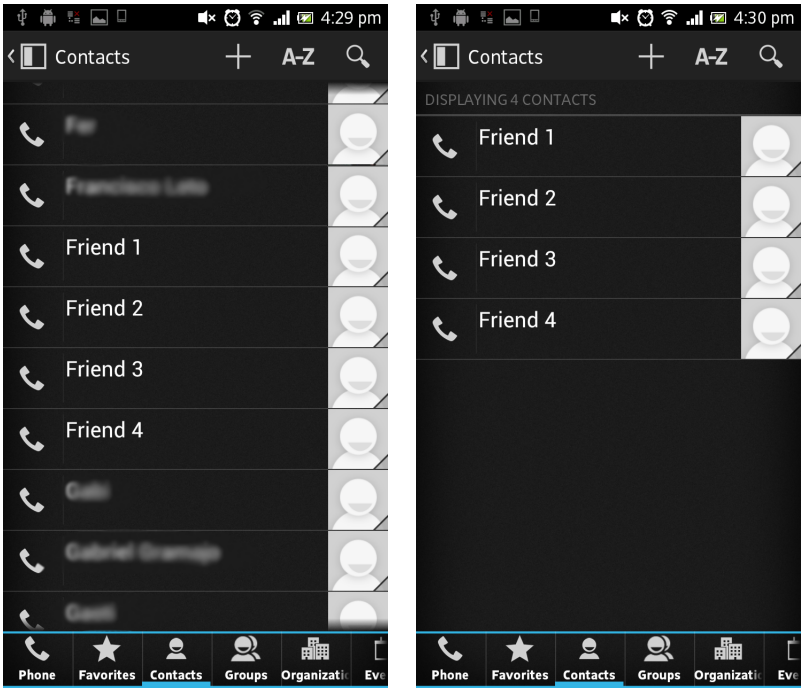
Como puede verse en la consola de logcat en la figura 24, *WhatsApp* llama siempre a la nueva versión de query y no a la vieja. Esto podría resolverse hookeando este nuevo método para las nuevas versiones. De forma más general, deberíamos adaptar los métodos hookeados de ASA según cambios de la API de Android. En futuras versiones, podría implementarse un analizador automático para detectar dichos cambios en la API de Android.

5.1.3. DW Contacts & Phone & Dialer

DW Contacts & Phone & Dialer es una aplicación que extiende al marcador de fábrica del dispositivo, además de manejar los contactos del mismo. Está focalizado en funcionalidades para empresas para mejorar la eficiencia durante el trabajo.

Utilizamos ASA para controlar el acceso al recurso de contactos de dicha aplicación con resultados exitosos.

Podemos observar en las figuras 25 como ASA controla el acceso a la agenda de contactos cuando definimos una política que sólo permita el acceso al grupo de *Amigos*.



(a) Todos los contactos (b) Grupo de Amigos

FIGURA 25. Controlando el acceso a contactos de *DW Contacts & Phone & Dialer* (Sony Xperia)

5.2. Datos del dispositivo

5.2.1. Android Device ID

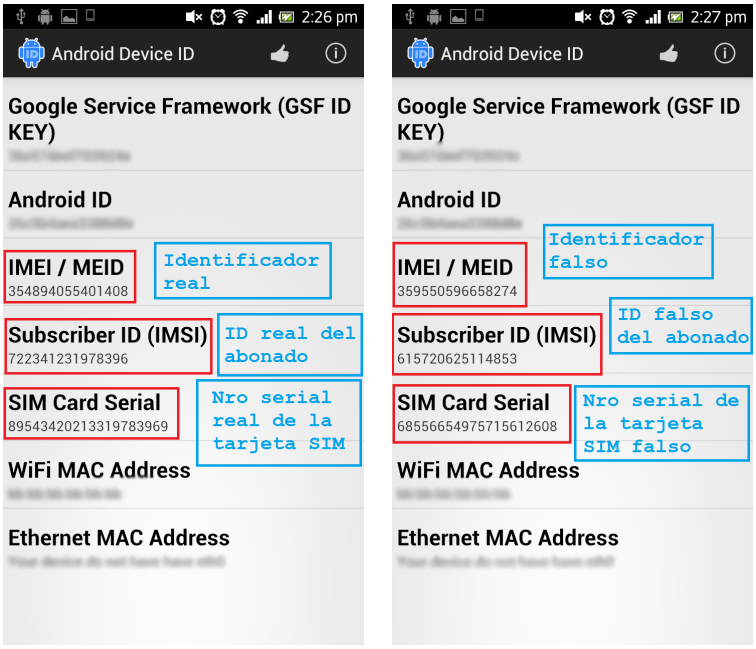
Android Device ID es una aplicación muy simple que nos permite obtener el IMEI o MEID, ID del abonado (IMSI), el número serial de la tarjeta SIM, la dirección MAC, entre otros datos. Se pueden copiar estos datos y compartir.
Se utilizó ASA para controlar esta aplicación utilizando por un lado valores falsos y por otro se configuró con valores aleatorios para mostrar que en dos instancias de la aplicación, los valores son efectivamente distintos.

En la figura 26 observamos los valores reales y falsos y en la figura 27 los dos conjuntos de datos aleatorios, según va cambiando la política definida por el usuario de ASA.

5.2.2. Know Your Phone

Know Your Phone es una herramienta rápida y simple que obtiene tu IMSI, IMEI, modelo de dispositivo, modelo del teléfono, versión de Android y número de teléfono (si éste es soportado).

Utilizamos ASA para modificar el acceso de esta aplicación para el Samsung Galaxy SIII. Dado que la API de Android no cambió particularmente



(a) Reales (b) Falsos

FIGURA 26. *Android Device ID* usando datos reales y falsos (Sony Xperia)

en el acceso a este recurso, ASA sigue siendo efectivo para Android 4.1.2. En la figura 28 podemos ver cómo la aplicación accede a los datos reales y, luego de cambiar la política a través de ASA, a datos falsos del equipo.

Finalmente, usamos ASA y su funcionalidad que permite editar los datos del teléfono para definir una nueva política para esta aplicación, como puede observarse en 29. En la primera figura vemos la ventana donde se edita la configuración de *Know Your Phone* y en la segunda, cómo estos nuevos valores tienen efecto en la aplicación.

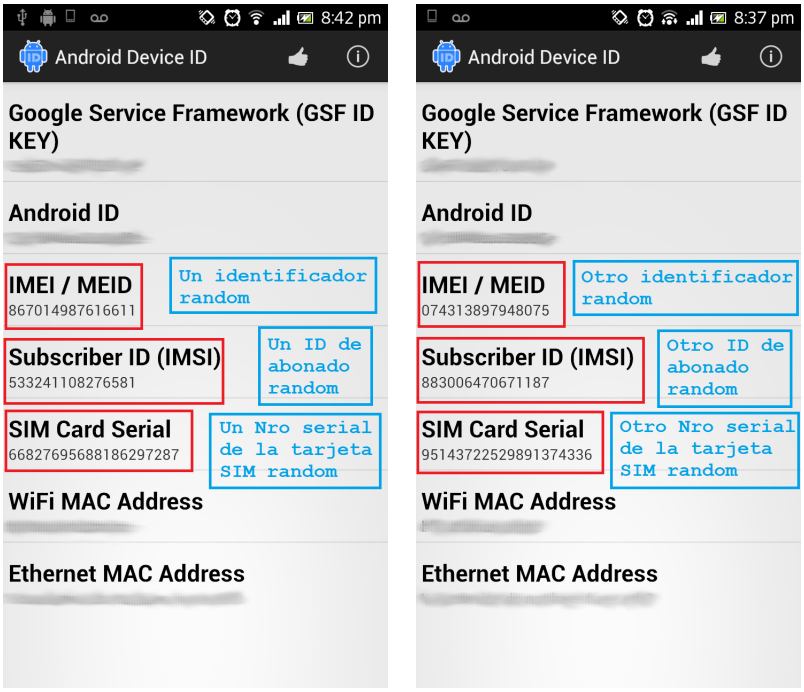
5.2.3. System Settings

Finalmente, nos pareció importante mostrar que ASA permite controlar el acceso a los datos del equipo, incluso para aplicaciones del sistema.

El único lugar donde el sistema muestra el identificador único del dispositivo es a través del menú en *Phone* → *Status*. En la figura 30 vemos cómo el sistema muestra el IMEI real para luego cambiar a uno falso, según la política definida en ASA.

5.2.4. Cómo el sistema toma los datos de mi dispositivo

En el caso del IMEI, el sistema usa una llamada que no está en la API pública de Android. Para poder probar esto, desensamblamos el código del



(a) Aleatorio 1 (b) Aleatorio 2

FIGURA 27. *Android Device ID* usando datos aleatorios más de una vez (Sony Xperia)

APK de la aplicación del sistema conocido como *Settings* (Settings.apk) para saber exactamente qué llamada hacía al momento de obtener el IMEI.

Primero obtuvimos el APK de *Settings*. Por lo general puede encontrarse en el directorio */system/data/Settings.apk* del dispositivo.

Luego utilizamos la herramienta *dex2jar* y su comando

```
d2j-dex2jar Settings.apk
```

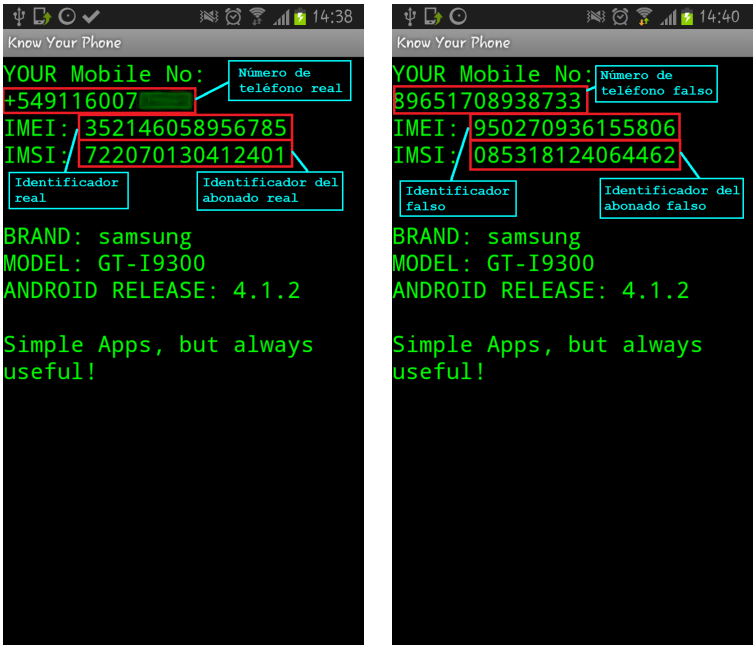
para obtener el *.jar* asociados al apk.

Finalmente, para poder ver el código fuente a partir del JAR obtenido usamos *Java Decompiler GUI*.
En la figura 31 vemos cómo una de todas las clases en el *Settings.apk* decompilado (en particular, la clase *Status.class*) importa y luego usa la clase *Phone* en el paquete *com.android.internal*.

5.3. Redes inalámbricas

5.3.1. Your Wifi

Your Wifi es una herramienta simple que permite ver todos los dispositivos conectados a una red. Provee información relevante como ser dirección



(a) Reales (b) Falsos

FIGURA 28. *Know Your Phone* usando datos reales y falsos (Samsung Galaxy SIII)

IP, fabricante, nombre del dispositivo y su dirección MAC.

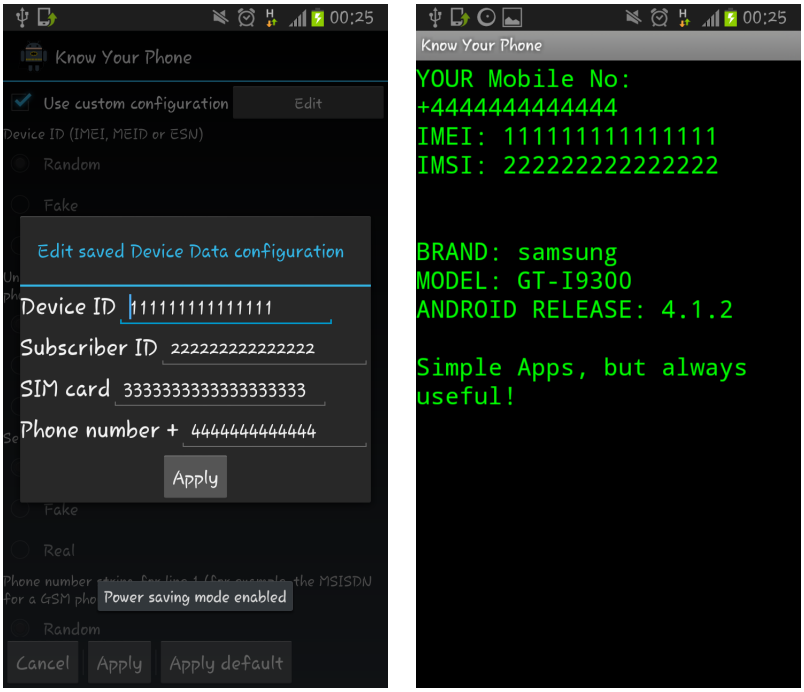
Usamos ASA para modificar los datos de redes inalámbricas de nuestro dispositivo detectados por esta aplicación.

Al momento de consultar los datos de la red inalámbrica actual del dispositivo, la aplicación recibe la información falsa según la política que definimos. Se realizaron pruebas definiendo distintas políticas y se determinó que, sin importar la configuración de ASA, la aplicación sigue usando los datos falsos, tanto cuando muestra el estado actual del dispositivo como cuando escanea todos los dispositivos conectados a la red actual.

En las figuras 32 y 33 podemos observar que ASA controla totalmente el acceso a este recurso por parte de *Your Wifi*.

5.3.2. Ping

PING es un acrónimo para *Packet Internet Groper*, que es algo así como *Buscador o rastreador de paquetes en redes*. Es una utilidad de diagnóstico en redes de computadoras que comprueba el estado de la comunicación con el host local con uno o varios equipos remotos de una red TCP/IP por medio del envío de paquetes ICMP de solicitud y de respuesta. Mediante esta utilidad se puede diagnosticar el estado, velocidad y calidad de una red



(a) Editando datos del dispositivo (b) Política *Custom* aplicada

FIGURA 29. *Know Your Phone* usando datos editados (Samsung Galaxy SIII)

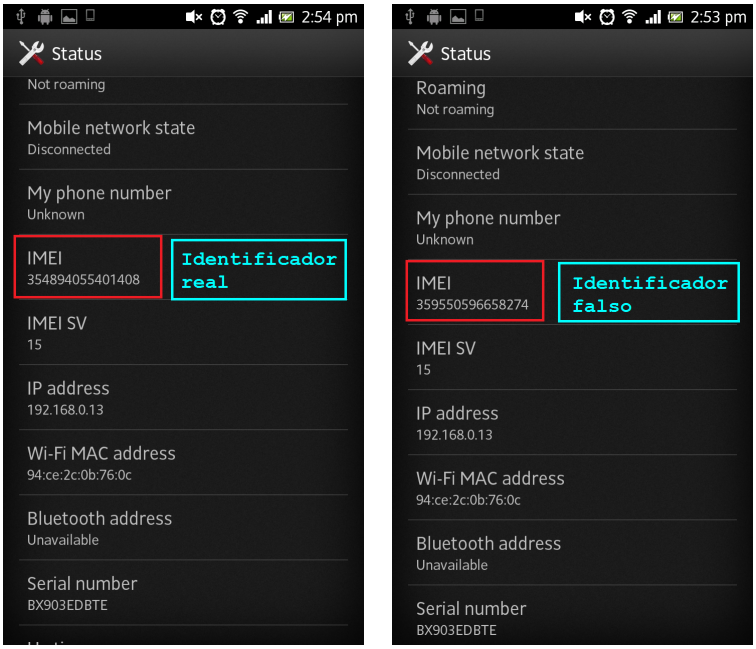
determinada.

Al realizar un ping contra la dirección IP verdadera de nuestro dispositivo, notamos que, a pesar de que la política para todas las aplicaciones es usar una dirección IP falsa, el dispositivo responde al pedido ICMP.
Esto era de esperarse, dado que ASA modifica sólo el acceso a la información en sí y no la configuración de la red inalámbrica.

En la figura 35 podemos ver que utilizando el IP real del dispositivo 192.168.0.13, el mismo responde y, como vimos en la sección anterior, habíamos modificado el mismo para que las aplicaciones usen uno falso 192.168.0.2.

5.3.3. Wireshark

De forma consistente con el punto anterior, cuando una aplicación va a acceder a los recursos de internet, la dirección de salida del paquete enviado por el dispositivo, se corresponde con el valor real de la IP, sin importar la política definida por ASA. Esto da evidencia de que ASA puede adulterar la información de la dirección IP sin comprometer la conectividad del equipo. La idea de ASA es proteger la privacidad del usuario sin modificar el recurso en sí.



(a) Reales (b) Falsos

FIGURA 30. *Settings* del sistema usando datos reales y falsos (Sony Xperia)

Es importante aclarar que una aplicación siempre puede obtener la dirección IP pública del dispositivo generando un pedido HTTP a un servidor. Sin embargo, ASA permite ocultar la dirección IP privada.

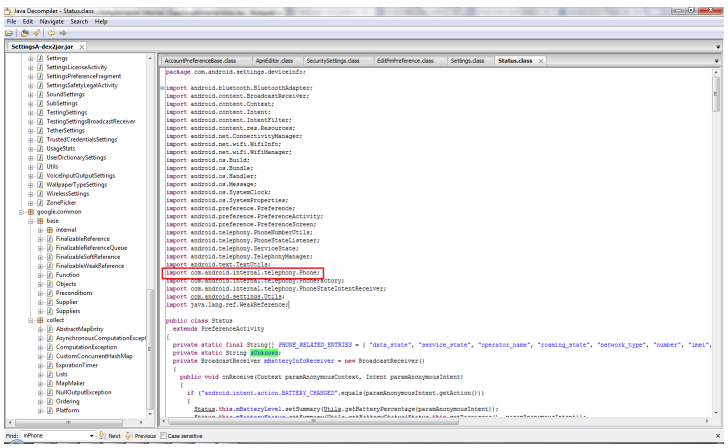
5.3.4. System Settings

Un dispositivo Android tiene una aplicación instalada de fábrica que por lo general se llama *System Settings* del inglés *Ajustes del sistema*, que permite configurarlo según las preferencias del usuario, además de proveer información básica del mismo.

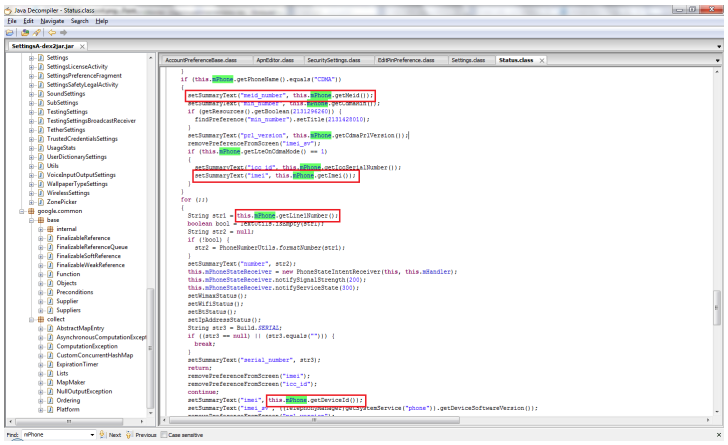
Hay dos formas de ver información de las redes inalámbricas usando esta aplicación. Una es a través del menú de *WiFi* → *Advanced* y otra es la que obtenemos a partir del menú *Phone* → *Status*. En las figuras 36 y 37 vemos estos dos menús y cómo ASA permite o no el acceso a los datos reales del equipo.

5.4. Geolocalización

Para analizar el comportamiento de ASA para el recurso de geolocalización, nos parece importante destacar que ASA funciona para cualquiera de los medios por los cuales una aplicación puede obtener nuestra ubicación.



(a) Importando la clase Phone

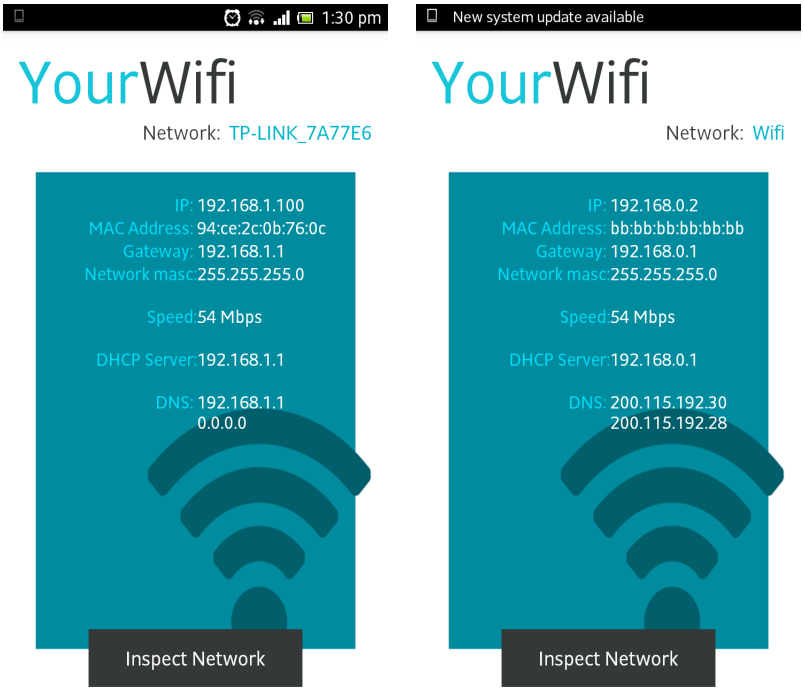


(b) Usando la clase Phone

FIGURA 31. *Settings* del sistema usando datos reales y falsos (Sony Xperia)

Por un lado podemos afirmar esto con pruebas realizadas usando ASA en los equipos antes nombrados, pero además, se realizó una búsqueda de aplicaciones disponibles en el *Google Play* que dicen poder simular una ubicación falsa del dispositivo para aplicaciones del mismo. Entre las más populares encontramos (aunque la lista es extensa):

- Fake GPS location (con más de 500,000 instalaciones en *Google Play*): en su descripción, tiene una sección que dice ser importante donde aclara que el usuario DEBE deshabilitar la opción *Use wireless networks* (Usar redes inalámbricas) en el menú *Location settings* del equipo para evitar que la aplicación no pueda mentir su ubicación una vez que se cierra la misma.
- Fake GPS Location Spoofer (con más de 5,000 instalaciones en *Google Play*): en su descripción pide en mayúsculas: *DISABLE WIFI LOCATION SETTINGS!!!!* (deshabilitar en ajustes de ubicación el WiFi).



(a) Reales (b) Falsos

FIGURA 32. *Your Wifi* usando datos reales y falsos (Sony Xperia)

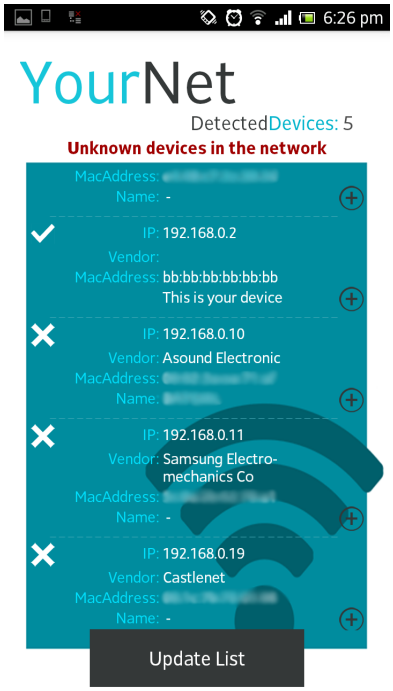


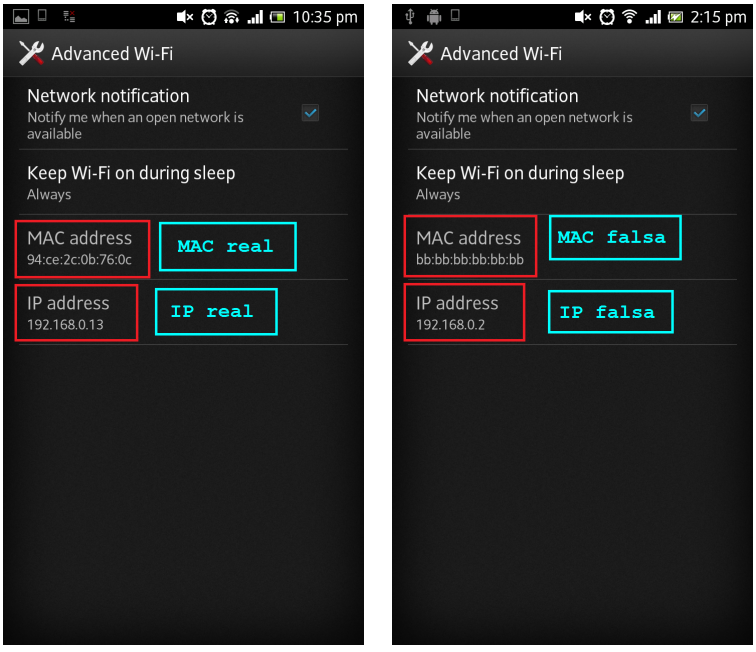
FIGURA 33. Escaneo de todos los dispositivos por *Your Wifi* (Sony Xperia)

```
~ > ping 192.168.0.13
PING 192.168.0.13 (192.168.0.13) 56(84) bytes of data.
64 bytes from 192.168.0.13: icmp_req=1 ttl=64 time=1095 ms
64 bytes from 192.168.0.13: icmp_req=2 ttl=64 time=96.0 ms
64 bytes from 192.168.0.13: icmp_req=3 ttl=64 time=15.6 ms
64 bytes from 192.168.0.13: icmp_req=4 ttl=64 time=267 ms
64 bytes from 192.168.0.13: icmp_req=5 ttl=64 time=60.3 ms
64 bytes from 192.168.0.13: icmp_req=6 ttl=64 time=305 ms
64 bytes from 192.168.0.13: icmp_req=7 ttl=64 time=105 ms
64 bytes from 192.168.0.13: icmp_req=8 ttl=64 time=26.3 ms
64 bytes from 192.168.0.13: icmp_req=9 ttl=64 time=263 ms
64 bytes from 192.168.0.13: icmp_req=10 ttl=64 time=71.7 ms
^C
--- 192.168.0.13 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9009ms
rtt min/avg/max/mdev = 15.671/230.720/1095.037/305.171 ms, pipe 2
```

FIGURA 34. Ping contra el IP real del dispositivo

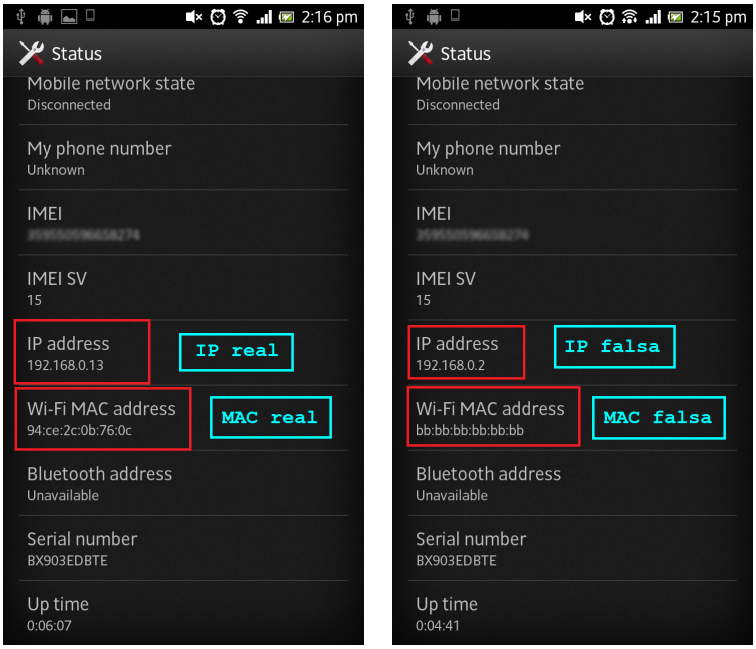
No.	Source	Destination	Protocol	Info
2859	192.168.0.10	192.168.0.22	ICMP	Echo (ping) request
2861	192.168.0.10	192.168.0.22	ICMP	Echo (ping) request
2863	192.168.0.10	192.168.0.22	ICMP	Echo (ping) request
2865	192.168.0.10	192.168.0.22	ICMP	Echo (ping) request
2869	192.168.0.10	192.168.0.22	ICMP	Echo (ping) request
2871	192.168.0.10	192.168.0.22	ICMP	Echo (ping) request

FIGURA 35. Ping desde el dispositivo a un equipo conectado a la misma red



(a) Reales (b) Falsos

FIGURA 36. *WiFi* → *Advanced* (Sony Xperia)



(a) Reales (b) Falsos

FIGURA 37. *Phone* → *Status* (Sony Xperia)

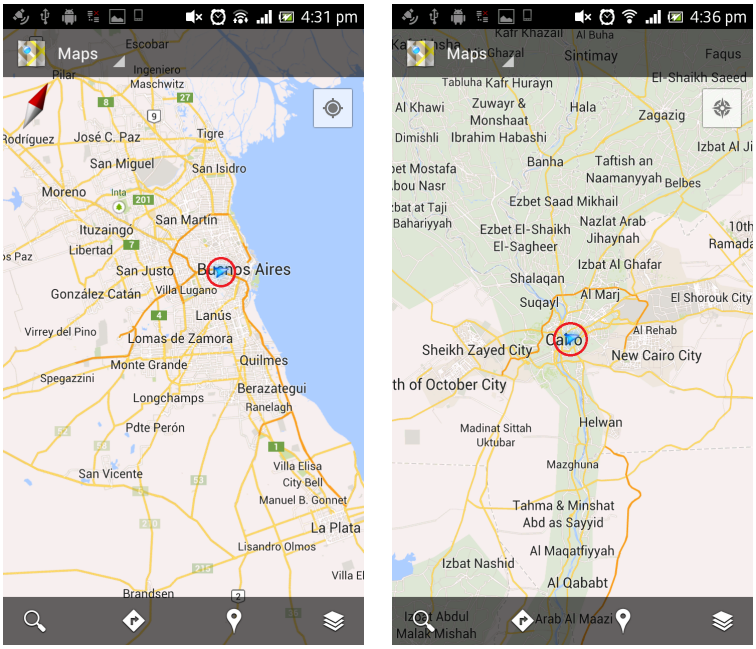
- Fake gps - fake location (con más de 10,000 instalaciones en *Google Play*): en su descripción piden que estemos seguros de tener la ubicación por WiFi apagada (*Make sure you have wifi location turned off*).

Es decir, ninguna funciona con todos los recursos del dispositivo prendidos, además de que piden tener la opción de desarrollo llamada *Mock Location* prendida, mientras ASA no tiene ninguno de estos requerimientos adicionales, aunque requiere permisos de *root*. Otro detalle que diferencia a ASA de las aplicaciones disponibles en *Google Play* es que permitimos definir una ubicación distinta por aplicación, mientras que las implementaciones existentes modifican la ubicación del equipo para todas las aplicaciones juntas.

5.4.1. Google Maps

Utilizamos ASA para mentir nuestra ubicación en *Google Maps*. Primero se utilizó una ubicación falsa marcando un punto en el mapa. En particular, en El Cairo. Esto puede observarse en la figura 38 donde vemos que a las 4:31 pm nos encontrábamos en la zona de Boedo para luego estar en El Cairo a las 4:36 pm.

Luego, utilizamos una ubicación que teníamos guardada previamente en Florencio Varela y la utilizamos para teletransportarnos de Florencio Varela (a las 00:00) a Boedo (a las 00.03). Esto puedo observarse en la figura 39.



(a) Ubicación Real (b) El Cairo

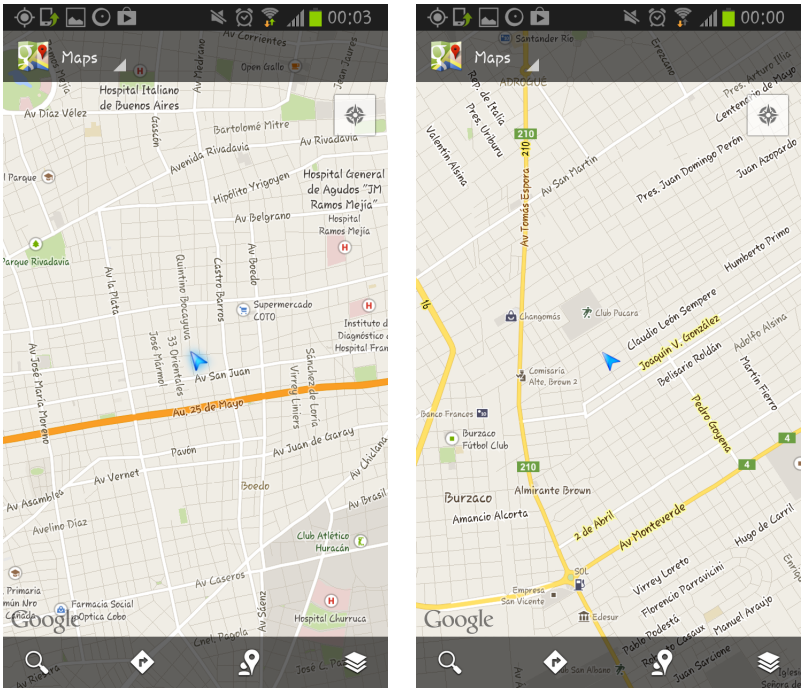
FIGURA 38. Controlando el acceso de *Google Maps* a nuestra ubicación (Sony Xperia)

5.4.2. Facebook

Facebook es la aplicación para dispositivos móviles de la red social más usada a nivel mundial en este momento con más de 500,000,000 instalaciones en *Google Play*.

Utilizamos ASA para controlar su acceso a nuestra ubicación. Primero utilizamos ASA marcando un punto en París y luego en EEUU y comprobamos que *Facebook* cree que estamos en cada uno de estos puntos, como puede verse en la figura 40. Notar que en el caso del control del recurso de geolocalización, ASA sigue funcionando correctamente para Android 4.1.2.

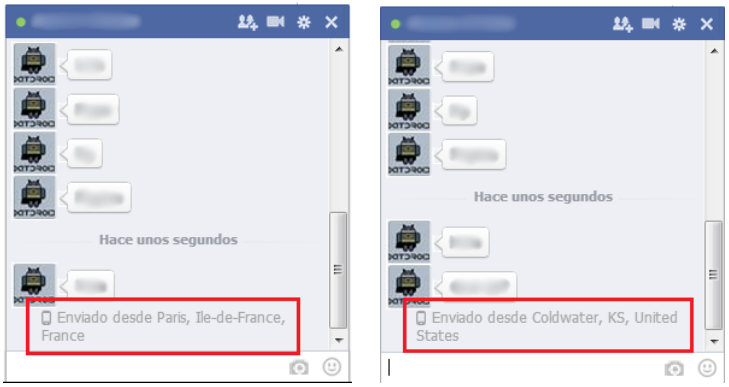
Luego, utilizamos ASA con la misma ubicación de Florencio Varela que usamos con *Google Maps* en la sección anterior y comprobamos que *Facebook* cree que estamos en Florencio Varela, para luego movernos a Boedo un instante después, como puede verse en la figura 41.



(a) Boedo: ubicación real

(b) Florencio Varela: ubicación guardada

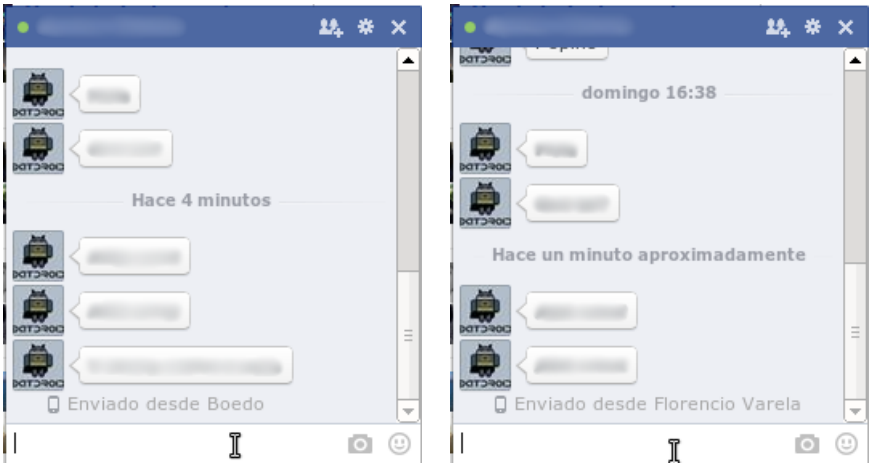
FIGURA 39. Controlando el acceso de *Google Maps* a nuestra ubicación (Sony Xperia)



(a) París

(b) EEUU

FIGURA 40. Controlando el acceso de *Facebook* a nuestra ubicación (Samsung Galaxy SIII)



(a) Boedo: ubicación real (b) Florencio Varela: ubicación guardada

FIGURA 41. Controlando el acceso de *Facebook* a nuestra ubicación (Samsung Galaxy SIII)

Capítulo 6

Conclusiones

En este trabajo presentamos ASA, un sistema que permite a un usuario del sistema operativo Android definir una política que le permita utilizar una aplicación con el adicional de poder controlar el acceso de la misma a un recurso sensible que exponga información que considere privada.

A lo largo de este informe se presentó el por qué el esquema de seguridad de Android, así como algunos trabajos ya implementados para extenderlo, no son suficientes desde el punto de vista del usuario y su privacidad. Desde el momento en el que se adquiere un dispositivo Android, o cuando se instala una aplicación nueva, depositamos un gran nivel confianza en el fabricante del dispositivo, la operadora telefónica elegida, así como el o los desarrolladores de cada aplicación que decidimos instalar. Dado que los mecanismos existentes en Android no son suficientes para prevenir posibles abusos de confianza por parte de éstos, vemos la necesidad de desarrollar un sistema que cubra dichos aspectos desde el punto de vista de distintos tipos de usuarios que pueden llegar a tomar diferentes decisiones dependiendo de la aplicación a controlar y el o los recursos que la misma requiere.

ASA permite a un usuario definir distintas políticas por aplicación y por recurso, así como políticas por defecto para un recurso dado y las aplicaciones del sistema o las de terceros como un conjunto. Este trabajo cubre un primer conjunto de recursos: la agenda de contactos, datos del dispositivo, información de las redes inalámbricas y geolocalización. Queda como tarea a futuro el dar soporte a otras versiones de Android, además de poder extender el alcance de ASA a otros recursos que comprometan la privacidad del usuario.

Este trabajo lleva consigo consideraciones legales que no fueron desarrolladas en el mismo, resultado de una posible violación en las condiciones de uso aceptadas al instalar una aplicación, y las implicaciones de dicha violación para ambos partes del acuerdo.

Es importante destacar que la idea principal de este trabajo es, por un lado, avanzar en la discusión sobre la problemática del abuso de confianza que compromete la privacidad de un usuario y exponer algunas ideas de cómo mitigarla. Por otro lado, introducir un sistema desarrollado para tal fin y compararlo con otros trabajos relacionados, pero teniendo en cuenta de que se trata de un sistema en evolución que planea seguir creciendo, recibiendo críticas y opiniones por parte de usuarios reales con distintos perfiles

que lo usen en situaciones de su vida cotidiana.

Capítulo 7

Trabajos futuros

En esta sección listaremos algunas de las ideas de cómo mejorar y extender este trabajo, que fueron surgiendo durante su desarrollo.

- Actualmente, un usuario debe tener en cuenta que las redes escaneadas deben mantener una coherencia entre la política de WiFi y la política de geolocalización al momento en que utiliza una configuración guardada. Por el momento, ASA sólo notifica al usuario que, cuando define una política de geolocalización, debe establecer como política de WiFi para dicha aplicación el devolver un conjunto de redes WiFi vacio o una política coherente de WiFi respecto a su ubicación. Un usuario que quiere utilizar las redes escaneadas de una ubicación particular para engañar una aplicación, puede no sólo guardar el estado actual de su ubicación sino también el de WiFi (que incluye las redes escaneadas) para poder utilizarlo como política de WiFi. De esta forma, proveedores como *NetworkProvider* podrían ubicar al dispositivo utilizando esta información de forma coherente.
  ASA podría extenderse para que cuando se guarde un estado de ubicación, automáticamente se guarden las redes escaneadas en ese momento, para que cuando dicha política sea utilizada, las redes WiFi guardadas sean empleadas sin intervención del usuario. Además, debería de eliminarse la opción de definir la política de las redes escaneadas en el recurso de WiFi y dejarle ese control al recurso de geolocalización.

- Agregar control de acceso a la tarjeta SD de forma tal que, para cada aplicación se pueda definir una política especificando las carpetas de la memoria externa a las que la misma puede acceder. Al momento en que una aplicación examine una carpeta a la que no tenga acceso, se le devolverá una carpeta vacía como resultado.

- Agregar soporte para IPv6 para el acceso de la información del IP del recurso de WiFi, dado que actualmente sólo soporta IPv4.

- Como se mencionó previamente, el proveedor *LocationManager* puede obtener la ubicación de un dispositivo mediante *requestUpdate*. Hay dos formas de hacerlo, utilizando *LocationListeners*, que es la que utilizan todas las aplicaciones analizadas que acceden a la ubicación, o a través de *PendingIntent*. En un trabajo futuro se debería extender ASA para cubrir el uso de *PendingIntent* a pesar de que

no sea utilizada frecuentemente para obtener una ubicación.

- Extender el control de acceso de ASA a otros recursos que son posibles blancos de aplicaciones que abusan de la confianza de un usuario. Entre ellos, la información de cuentas de usuario, el historial de SMSs, el log de llamadas, el historial de navegación, etcétera.
  Un ejemplo puntual de este tipo de abusos es que una aplicación puede obtener el número de línea de un dispositivo, sin necesidad de contar con acceso explícito al permiso de *READ_PHONE_STATE*, a través de aplicaciones como *WhatsApp* que utilizan esta información como el nombre de usuario de su cuenta.

Bibliografía

[1] *Adrienne Porter Felt*, *Erika Chin*, *Steve Hanna*, *Dawn Song*, *David Wagner Android Permissions Demystified*.

[2] *Alastair R. Beresford*, *Andrew Rice*, *Ripduman Sohan* y *Nicholas Skehin*, *MockDroid: Trading Privacy for Application Functionality on Smartphones*.

[3] *David Wetherall*, *Stuart Schechtery*, *Seungyeop Han*, *Peter Hornyack* y *Jaeyeon Jung*, *"These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications, aka AppFence*.

[4] *Jinseong Jeon*, *Kristopher K. Micinski* y *Jeffrey A. Vaughan*, *Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications*.

[5] *Yajin Zhou,Xinwen Zhang,Xuxian Jiang* y *Vincent W. Freeh Taming Information-Stealing Smartphone Applications (on Android), aka TISSA*.

[6] *Michael Backes*, *Sebastian Gerling*, *Matteo Maffe*, *Philipp von Styp-Rekowsky* y *Christian Hammer*, *AppGuard — Real-time policy enforcement for third-party applications*.

[7] *Rubin Xu*, *Hassen Sa Di* y *Ross Anderson*, *Aurasium: Practical Policy Enforcement for Android Applications*.

[8] *LocationListener* `http://developer.android.com/reference/android/location/LocationListener.html`.

[9] *Cydia Substrate* `http://www.cydiasubstrate.com/api/java/`.

[10] *Android Platform Security Architecture* `http://source.android.com/devices/tech/security/`.

[11] *La linterna para Android que robaba datos de usuarios* `http://goo.gl/vEyB4N`.

[12] *Permission Manager* `https://play.google.com/store/apps/details?id=com.gmail.permissionmanager&hl=en`.

[13] *How Google -and everyone else- gets Wi-Fi location data* `http://www.zdnet.com/blog/networking/how-google-and-everyone-else-gets-wi-fi-location-data/1664`.

[14] *Processes and Threads* `http://developer.android.com/guide/components/processes-and-threads.html#ThreadSafe`.

[15] *Provider Implementation Notes* `https://thenewcircle.com/s/post/1375/android_content_provider_tutorial`.

[16] *Understanding Android: Three Ways to Find Your Location* `http://www.idyllramblings.com/2012/03/understanding-android-three-ways-to-find-your-location.html`.

[17] *RFC 2396* `http://www.ietf.org/rfc/rfc2396.txt`.

[18] *Precisión de ubicación* `https://support.google.com/gmm/answer/81873?hl=en`

[19] *ContentResolver* `http://developer.android.com/reference/android/content/ContentResolver.html`.

[20] *Empezar un servicio en un proceso nuevo* `http://developer.android.com/guide/topics/manifest/service-element.html`.

[21] *Diferentes procesos para una misma app* `http://developer.android.com/guide/components/processes-and-threads.html`.

[22] *Versiones de Android* `http://grepcode.com/project/repository.grepcode.com/java/ext/com.google.android/android/`.

[23] *Documentación de grupos* `http://developer.android.com/guide/topics/providers/contacts-provider.html#Groups`.

[24] *SQLiteDatabasse* http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/4.0.3_r1/android/database/sqlite/SQLiteDatabase.java#SQLiteDatabase.rawQuery%28java.lang.String%2Cjava.lang.String%5B%5D%29.

Documentación SQLiteDatabase http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html.

[25] *Michael Kern* y *Johannes Sametinger*, *Tracking in Android* UBICOMM 2012 : The Sixth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies.

Capítulo 8

Apéndices

8.1. Accediendo a las bases de datos de Android

Cuando trabajamos en una aplicación de Android que almacena datos en una base de datos *SQLite* surgen varias preguntas como ¿Dónde están almacenados en el filesystem estos archivos de base de datos? ¿Cómo puedo acceder a estas bases de datos? Al momento de detectar qué bases de datos *SQLite* eran modificadas por las aplicaciones que queremos controlar, por ejemplo, para detectar qué cambios son realizados en la base de datos de contactos cuando agregamos o modificamos un contacto en nuestro dispositivo, surge la necesidad de poder acceder e inspeccionar esas bases de datos.

Desde la ventana de *File Explorer* de Eclipse podemos ver las bases de datos pero no podemos ver la estructura de las tablas y sus datos.

Para esto utilizamos un plugin de Eclipse llamado *Eclipse Plugin SQlite Manager* que nos permite inspeccionar, siempre y cuando tengamos los permisos necesarios, las bases de datos de contactos e incluso, las bases de datos locales de cada aplicación, por ejemplo, la base de datos de WhatsApp *wa_contacts.db*.

De esta forma podemos ver la estructura de *wa_contacts*:

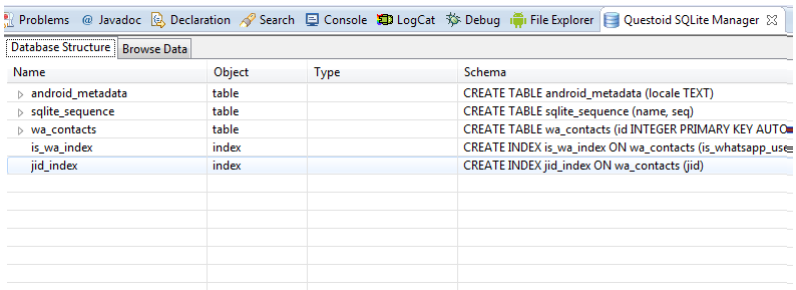


FIGURA 42. Estructura de la base de datos wacontacts

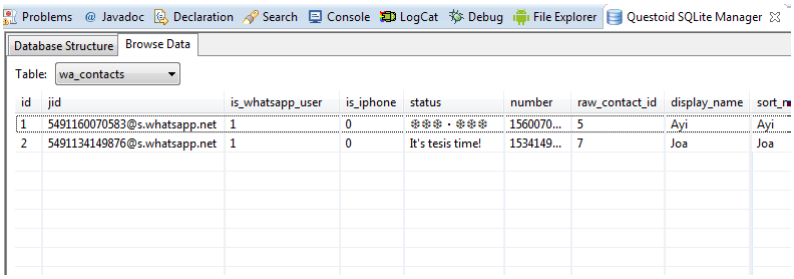


FIGURA 43. Contenido de la base de datos wacontacts