PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 13-08

Descripción y análisis del modelo

de seguridad de Android

Agustín Romano Carlos Luna

2013

Descripción y análisis del modelo de seguridad de Android

Agustín Romano, Carlos Luna

ISSN 0797-6410
Reporte Técnico RT 13-08
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, 2013

Descripción y análisis del modelo de seguridad de Android

Agustín Romano[1] y Carlos Luna[2]

[1]Universidad Nacional de Rosario, Rosario, Argentina
aromano@fceia.unr.edu.ar
[2]InCo, Facultad de Ingeniería, Universidad de la República, Uruguay
cluna@fing.edu.uy

Resumen

En los últimos años se ha observado un marcado incremento en el número de dispositivos móviles que tienen a Android como sistema operativo. Por este motivo, una falla en la seguridad de dicha plataforma afectaría a una gran cantidad de usuarios de distintas formas, por ejemplo, dejando expuesta información privilegiada guardada en el dispositivo. Este elevado número de víctimas potenciales alientan a los creadores de aplicaciones maliciosas a elegir a Android como objetivo de sus ataques. Es por ello que el análisis y fortalecimiento de su modelo de seguridad se ha convertido en una tarea importante que despertó el interés de numerosos investigadores. El objetivo de este trabajo es realizar una descripción exhaustiva del modelo de seguridad implementado por Android, incluyendo los análisis y críticas de los trabajos más relevantes hasta el momento y una comparación con el modelo de seguridad de los dispositivos móviles Java (JME-MIDP). De esta forma, el presente trabajo puede ser utilizado como disparador a la hora de comenzar a realizar estudios en áreas muy diversas sobre la seguridad de Android.

Palabras Clave. Dispositivos Móviles, Seguridad, Android, MIDP.

1. Introducción a Android

Android es un sistema operativo *open-source* [14] diseñado, inicialmente, para dispositivos móviles y desarrollado por Google junto con la Open Handset Alliance (OHA) [36]. Al instalar una distribución Android en un dispositivo móvil se incluye, además de un sistema operativo de base, un conjunto de aplicaciones principales (por ejemplo, libreta de contactos, reloj, etc.) que proveen acceso a las funcionalidades básicas del dispositivo. Adicionalmente, se cuenta con un *middleware* que ofrece librerías y servicios del sistema, tanto a las aplicaciones principales como a las instaladas posteriormente. En relación a estas

últimas, "el llamado Android Software Development Kit (SDK) incluye las herramientas necesarias para el desarrollo de nuevas aplicaciones en la plataforma Android usando el lenguaje de programación Java" [34].

Una de las características principales de Android es que cualquier aplicación, ya sea principal o creada por algún desarrollador, puede, al instalarse con las autorizaciones adecuadas, utilizar tanto los recursos/servicios del dispositivo móvil (hacer llamadas, leer libreta de contactos, etc.) como los ofrecidos por el resto de las aplicaciones instaladas.

A lo largo de esta sección explicaremos las características principales de Android, incluyendo su arquitectura y conceptos básicos. El resto del trabajo se organiza de la siguiente forma: la Sección 2 ofrece una descripción exhaustiva del modelo de seguridad en Android, en la Sección 3 se resume el estado del arte con respecto al análisis de dicho modelo, describiendo los trabajos más relevantes en este campo. Luego, en la Sección 4 se realiza un análisis comparativo entre el modelo de seguridad de Android y el de los dispositivos móviles Java (JME-MIDP) y, finalmente, la Sección 5 presenta las conclusiones del reporte y los trabajos futuros.

1.1. Arquitectura

La arquitectura del sistema operativo Android sigue el estilo arquitectónico conocido como Sistemas Estratificados, ya que los distintos componentes del mismo son divididos en *estratos* que conforman una jerarquía con respecto al nivel de abstracción. Mientras que los estratos más bajos agrupan componentes ligados a la interacción con el hardware del dispositivo, los estratos superiores se corresponden con tareas de más alto nivel. En líneas generales, los componentes de un estrato determinado utilizan los servicios provistos por el estrato inferior (de existir) y ofrecen sus propios servicios a los componentes del estrato superior [25, 41].

Los distintos estratos en la arquitectura de Android se describen en la Figura 1 y, a lo largo de las secciones subsiguientes, se resumen las principales características de cada uno de ellos.

1.1.1. Kernel Linux

"Este estrato funciona como una capa de abstracción entre el hardware y el resto de los componentes del sistema" [34] proveyendo a estos últimos una interfaz para acceder a todos los recursos del dispositivo móvil. Las funcionalidades que provee este estrato dependen de un kernel Linux multiusuario que se encarga, entre otras cosas, de la administración de la memoria, los procesos y los drivers de los distintos recursos. Dicho kernel también implementa aspectos básicos del modelo de seguridad en Android. Como consecuencia, por ser éste el primer estrato en la arquitectura, la base del sistema operativo Android es un kernel Linux y, por lo tanto, las características básicas del primero dependerán en gran medida del segundo.

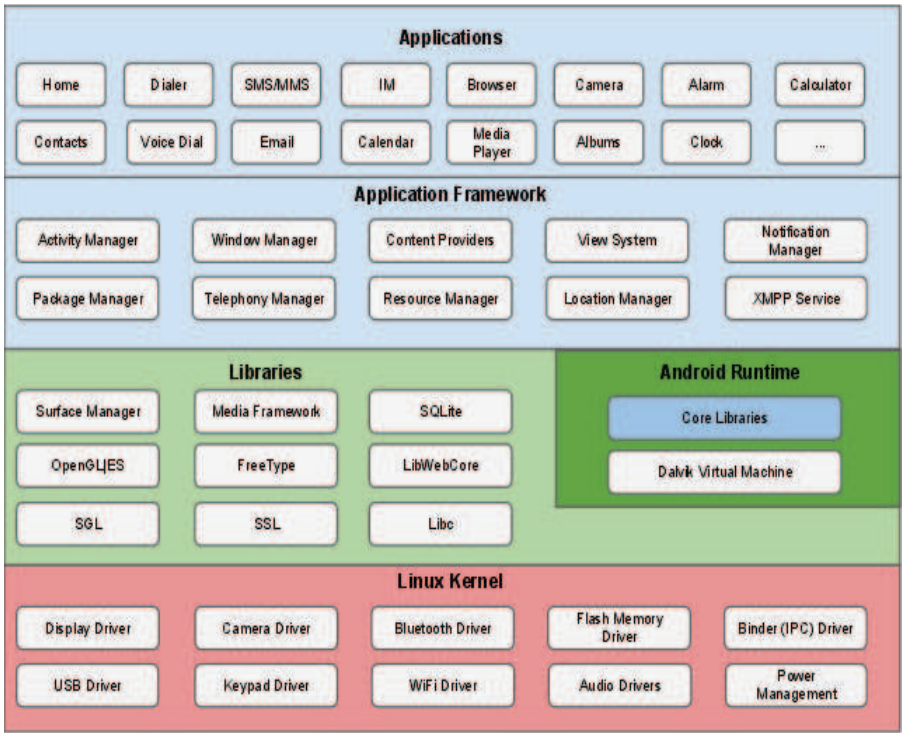

Figura 1: Descripción de la arquitectura de Android publicada en *Android Open Source Project* [13].

1.1.2. Librerías y tiempo de ejecución

Este estrato agrupa, básicamente, tres tipos de componentes:

- *Librerías nativas.* Se ofrece un conjunto de librerías C/C++ que proveen algunos servicios básicos para aplicaciones y otros programas. Dichos servicios son utilizados por los componentes del estrato superior. Entre sus funcionalidades se incluye facilitar el acceso al hardware (por ejemplo, manejo de gráficos) y a bases de datos. Estas librerías nativas corren en distintos procesos del kernel Linux subyacente [34, 41].

- *Máquina virtual.* Android cuenta con una máquina virtual, llamada Dalvik, para la ejecución de las aplicaciones programadas en Java. Ésta "ejecuta archivos con formato *Dalvik Executable* (.dex) que está optimizado para reducir al mínimo el consumo de memoria del dispositivo móvil" [34]. Cada aplicación Java es compilada a un formato *bytecode* y, posteriormente, es ejecutada en una máquina virtual Dalvik propia, distinta a la asignada a cualquier otra aplicación. Además de utilizar el lenguaje de programación Java en el desarrollo de una aplicación, también es posible

incluir código nativo (como C o C++) que corra por fuera de la máquina virtual Dalvik. Este tipo de código, al compilarse, se ejecuta directamente en el procesador del dispositivo móvil. Sin embargo, la ejecución de dicho código nativo continúa siendo afectada por las restricciones del kernel Linux subyacente [2, 41].

- *Librerías Estándar.* Se provee, a nivel de ejecución de aplicaciones, "la mayoría de las funcionalidades disponibles en las librerías estándar de Java, como por ejemplo, operaciones matemáticas, de texto, de entrada/salida, entre otras" [34].

1.1.3. Framework de aplicaciones

En este estrato se encuentra el framework que se encarga de, entre otras cosas, administrar el ciclo de vida de cada aplicación, proveer el conjunto de APIs necesarias para el desarrollo y manejar la interacción tanto entre aplicaciones como también entre las distintas partes que las componen. Las características de este framework imponen una estructura determinada para las aplicaciones que, en algunos aspectos, difieren de los programas tradicionales (por ejemplo, en las aplicaciones Android "no existe una función `main` o *entry point* único para la ejecución" [27]).

Uno de los objetivos principales de la arquitectura de Android es la reutilización de componentes: cualquier aplicación puede ofrecer sus servicios y cualquier otra, con la autorización correspondiente, puede usarlos [34]. Para esto, Android permite que la comunicación entre aplicaciones se realice con un gran nivel de granularidad, posibilitando la interacción directa entre los distintos componentes de las aplicaciones.

"Los desarrolladores tienen acceso a las mismas APIs que utilizan las aplicaciones principales" [34] (es decir, las aplicaciones que se encuentran preinstaladas). Todas las librerías que implementan las APIs ofrecidas por este framework están escritas en el lenguaje de programación Java y residen en la máquina virtual Dalvik de cada aplicación. Al mismo tiempo, para llevar a cabo alguna acción solicitada por una aplicación, estas librerías se comunican con el sistema Linux de base en donde, además, se verifica si la aplicación solicitante tiene los permisos necesarios para acceder a los recursos del sistema o de otras aplicaciones que estén involucradas en la acción en cuestión [28, 34]. Dado que el sistema de permisos es una parte esencial del modelo de seguridad de Android, se brindarán más detalles con respecto al mismo en las secciones subsiguientes.

1.1.4. Aplicaciones

Finalmente, en este estrato se incluyen tanto las aplicaciones principales (por ejemplo, cliente de e-mail, calendario, libreta de contactos) como las nuevas aplicaciones escritas por otros desarrolladores [34]. Ambos tipos de aplicaciones se ejecutan en el marco impuesto por el framework del estrato inferior. Sin embargo, el acceso a las APIs está restringido a los fragmentos de aplicaciones escritos en el lenguaje de programación Java. De existir código nativo, no se

podrá acceder directamente a las APIs a través del mismo sin antes interponer código Java que actúe como intermediario [28].

1.2. Componentes de una aplicación

Como se adelantó en la sección anterior, uno de los lineamientos principales que impone el framework de Android es la estructura de las aplicaciones. Una aplicación Android se construye a partir de distintos bloques básicos llamados *componentes*[1]. Cada componente "existe como una entidad única y cumple un rol específico, definiendo así, entre todos ellos, el comportamiento general de la aplicación" [3]. Un aspecto característico del diseño del sistema Android es que una aplicación puede iniciar cualquier componente de otra, si se tienen los permisos adecuados.

"Existen cuatro tipos de componentes de una aplicación. Cada tipo se utiliza para un propósito diferente y tiene un ciclo de vida distinto" [3]. Estos tipos son:

Actividades

Una actividad representa una pantalla de la aplicación, en donde se provee una interfaz de usuario para interactuar con la misma [3]. Típicamente, cada aplicación tiene una actividad principal que representa la primer pantalla que ve el usuario al ser iniciada desde la lista de aplicaciones disponibles. A partir de ese momento, es posible pasar a la siguiente pantalla (de existir) llamando a una nueva actividad [1, 11]. A pesar de que una aplicación tenga una actividad principal, cada una de éstas es independiente de las otras, dando la posibilidad a una aplicación distinta de iniciar cualquier actividad que no sea, necesariamente, la principal (siempre y cuando se tenga la autorización apropiada). Por ejemplo, podría existir una aplicación que funcione como cliente de correo electrónico y que consista en tres pantallas/actividades que se comuniquen entre sí: bandeja de entrada, nuevo correo y elementos enviados. A pesar de que la actividad principal podría ser, típicamente, la bandeja de entrada, cualquier otra aplicación que desee permitir a su usuario mandar un e-mail podría acceder directamente a la actividad 'nuevo correo' de la aplicación original, utilizando los servicios de la misma sin tener que pasar por la bandeja de entrada [3]. Esta posibilidad de reutilizar componentes es una de las características principales del sistema Android.

Servicios

Un servicio es un componente cuya ejecución se desarrolla en segundo plano sin ofrecer ninguna interfaz con el usuario. Cualquier componente con los permisos adecuados puede iniciar un servicio o asociarse a uno en ejecución para interactuar con él [3].

[1] En lo que resta del trabajo, sólo utilizaremos este nombre para referirnos a dichos bloques básicos.

El uso más frecuente de los servicios es para realizar tareas que demanden una gran cantidad de tiempo (y no requieran interacción con el usuario); sin embargo, también pueden ser utilizados con el fin de trabajar para procesos remotos. Por ejemplo, un servicio podría reproducir música o descargar un archivo a través de internet sin impedir que el usuario siga utilizando su teléfono móvil normalmente [3].

Content Providers

"Un *content provider* es un componente diseñado para compartir información entre aplicaciones. Dicha información puede estar guardada, por ejemplo, en bases de datos SQLite, en la web o en cualquier otro medio de almacenamiento persistente que esté disponible" [3]. De esta forma, un *content provider* actúa como una interfaz entre los datos persistidos y el resto de las aplicaciones para que, estas últimas, puedan tanto acceder a los mismos como también modificarlos [41].

Existen dos formatos posibles en los que un *content provider* puede presentar la información almacenada internamente al resto de las aplicaciones. Estos son: archivos y tablas. Por ejemplo, si se desea compartir el contenido de una base de datos SQLite, se optará por presentar dicha información en formato de tablas. Por su parte, el resto de las aplicaciones pueden acceder a la información ofrecida realizando consultas o, más directamente, mediante el uso de URIs (*Uniform Resource Identifiers*) que identifican a cada recurso del *content provider*. Dichos URIs tienen la siguiente forma:

```
content://authority_name/path/id
```

Donde `authority_name` es un nombre simbólico que identifica al *content provider*, `path` es un nombre que apunta a una tabla o archivo, dependiendo del formato elegido para presentar la información, y, como campo opcional, `id` apunta a una tabla o fila individual [4, 41].

Broadcast Receivers

Un *broadcast receiver* es un componente cuyo objetivo es recibir mensajes/anuncios, emitidos por el sistema u otra aplicación, y disparar acciones a partir de los mismos. Dichos mensajes, llamados *broadcasts*, se transmiten a lo largo de todo el sistema; y son los *broadcast receivers* los encargados de decidir cuáles de ellos se comunicarán a la aplicación a la que pertenecen. Por ejemplo, el sistema podría generar *broadcasts* cuando la batería esté por agotarse o cuando se tome una foto. Dependiendo de la aplicación, se configurarán sus respectivos *broadcast receivers* para suscribirse o no a esa clase de mensajes y efectuar, en caso de suscribirse, una acción determinada al recibirlos. Por su parte, una aplicación podría generar un *broadcast* cuando, por ejemplo, termine de descargar un archivo [3].

Por último, al recibir un mensaje, un *broadcast receiver* debe realizar operaciones de corta duración. Si se necesita realizar alguna operación que demande

una gran cantidad de tiempo, entonces el *broadcast receiver* puede, por ejemplo, iniciar un servicio para ello [3].

1.3. Interacción entre componentes

"Tres de los cuatro tipos de componentes, actividades, servicios y *broadcast receivers*, son activados mediante un mensaje asíncrono llamado *intent*". Estos mensajes hacen que distintos componentes individuales, pertenecientes tanto a una misma aplicación como a aplicaciones distintas, se relacionen entre sí en tiempo de ejecución [3]. Existen, principalmente, dos formas de utilizar un *intent*: como un *broadcast* o como un mensaje para interactuar con actividades y servicios.

Al crear un *intent* se pueden incluir los siguientes campos [5]:

- **componente destino:** Especifica el nombre del componente al que va dirigido el *intent*. De no especificarse ningún destinatario, a la hora de enviar el *intent*, el sistema debe elegirlo basándose en la información contenida en otros campos.

- **acción:** Describe la acción que se quiere realizar con el *intent* o, en el caso de ser utilizado como *broadcast*, el evento que sucedió. Algunas constantes predefinidas para este campo son: `ACTION_CALL`, para iniciar una llamada telefónica desde una actividad; `ACTION_EDIT`, para enviar información con el fin de ser editada en una actividad; o `ACTION_BATTERY_LOW`, para comunicar que la batería se está agotando. Adicionalmente, se pueden agregar constantes definidas por el usuario. Una vez elegido el valor de este campo, la estructura del resto del *intent* dependerá fuertemente de dicha elección.

- **categoría:** Especifica, con una o varias constantes, las características del componente que debería recibir el *intent*. Por ejemplo, la contante predefinida `CATEGORY_LAUNCHER` indica que el componente destino debería ser la actividad principal de una aplicación.

- **datos:** Incluye el o los URIs que identifican los datos sobre los cuales se quiere actuar. Dichos URIs pueden ser referencias a recursos de algún *content provider* o a otro tipo de datos. Por ejemplo, si al campo **acción** se le asigna el valor `ACTION_CALL`, se incluirá el URI del número de teléfono que se quiere llamar.

- **banderas:** Incluye banderas opcionales para realizar tareas específicas, como por ejemplo la delegación de permisos (ver Sección 2.4).

- **extras:** Ofrece información adicional.

Luego de crear el *intent*, éste es pasado como argumento a algún método que complete la acción deseada y que, además, envíe dicho *intent* al destinatario correspondiente (por ejemplo, el método `startActivity()` inicia la actividad especificada como destinatario por el *intent* que toma como argumento). Si se

da un valor para el campo **componente destino** de un *intent*, éste es enviado directamente al componente especificado; en caso contrario, el sistema debe resolver el destinatario en tiempo de ejecución. A cada tipo de *intent* se lo denomina, respectivamente, explícito e implícito. Con respecto a los *intents* implícitos, cualquier aplicación puede registrar algunos de sus componentes para que reciban *intents* con determinados valores para los campos **acción**, **categoría** o **datos**. Por ejemplo, una actividad con la capacidad de realizar llamadas telefónicas puede registrarse para recibir *intents* implícitos cuyo campo **acción** tenga el valor `ACTION_CALL`. En tiempo de ejecución, cuando el sistema tiene que resolver el destinatario de un *intent* implícito, se comparan los valores de los campos mencionados y se eligen como posibles destinatarios a los componentes registrados para recibir el tipo de *intent* en cuestión. En el caso de existir muchas actividades registradas para recibir el mismo *intent*, el sistema le solicita al usuario que elija una (a menos que ya exista una actividad por defecto para manejar este tipo de *intents*). En cambio, si los componentes registrados fueran servicios, el sistema elige de forma aleatoria entre los candidatos [23]. Por último, si los componentes habilitados para recibir el *intent* fueran *broadcast receivers*, el *intent* es enviado a todos ellos. Siguiendo el ejemplo de la sección anterior, si se está en la pantalla 'bandeja de entrada' del cliente de correo electrónico y se quiere redactar un nuevo e-mail, la actividad que representa a la pantalla actual deberá enviar un *intent* explícito para activar la actividad/pantalla correspondiente a 'nuevo correo'. Sin embargo, si se quiere visualizar una imagen adjunta, la actividad en ejecución podría enviar un *intent* implícito para que el usuario elija una aplicación acorde para dicha tarea.

1.4. *Android Manifest*

Para que el sistema pueda iniciar un componente de una aplicación, se debe dar a conocer la existencia del mismo. Para esto, toda aplicación Android debe incluir en su directorio raíz un archivo XML llamado `AndroidManifest`. En este archivo se declaran todos los componentes que forman parte de la aplicación en cuestión, junto con algunas características de los mismos que se definen de forma estática. Además, se identifican los permisos que requiere la aplicación para poder ejecutarse, como por ejemplo, acceso a internet o a la libreta de contactos [3]. Este último punto se desarrollará con más detalle en las próximas secciones.

En la Figura 2 se da un ejemplo de un archivo `AndroidManifest`.

2. El modelo de seguridad de Android

La implementación del modelo de seguridad de Android se lleva a cabo a lo largo de toda la arquitectura del sistema. A continuación detallamos los aspectos más importantes de dicho modelo.

```
<manifest package="com.example.project"... >
    <uses-permission android:name="android.permission.SET_WALLPAPER" />
    <application...>
        <activity android:name="com.example.project.OneActivity"... >
            ⋮
        </activity>
        <service android:name="com.example.project.OneService"... >
            ⋮
        </service>
        ⋮
    </application>
</manifest>
```

Figura 2: Ejemplo de un archivo `AndroidManifest` [12].

2.1. *Application Sandbox*

Android implementa el principio de mínimo privilegio haciendo que cada aplicación se ejecute en, como se lo denomina, un *sandbox* (palabra inglesa para arenero); es decir, forzando a que cada aplicación sólo pueda tener acceso irrestricto a sus propios recursos. Por defecto, ninguna aplicación puede acceder a otras partes del sistema (por ejemplo, cámara de fotos del teléfono móvil, libreta de contactos, otra aplicación); para ello se debe obtener el permiso correspondiente [3].

El mecanismo de *Application Sandbox* es implementado a nivel de kernel de la siguiente forma: a cada aplicación instalada se le asigna, excepto en casos especiales (ver Sección 2.2), un identificador de usuario (UID) distinto con pocos privilegios [28]. Además, se configuran todos los archivos creados en el almacenamiento interno para que, por defecto, sólo puedan ser accedidos por el UID de la aplicación que los creó. Sin embargo, también es posible configurar explícitamente un archivo para que pueda ser leído/escrito por cualquier otra aplicación. Los archivos creados en almacenamiento externo, como tarjetas SD, pueden ser leídos y modificados por cualquier aplicación con la sola restricción de tener el permiso adecuado para escribir en almacenamiento externo, de necesitar hacer-

lo[2]. Adicionalmente, se ejecuta cada aplicación como un proceso separado del resto, con su propio espacio de direcciones [3]. Debido a esto, valiéndose de los mecanismos de protección de los sistemas Linux, ninguna aplicación puede, al menos por defecto, acceder a los recursos de otra (ya que tienen distintos UIDs) ni a los recursos del sistema (ya que el UID asignado a cada aplicación tiene pocos privilegios[3]) y, dado que cada aplicación corre en un proceso diferente, la única forma de interacción entre ellas es a través de mecanismos de IPC (*Inter Process Communication*) [15]. Si bien el kernel provee los mecanismos de IPC tradicionales de Unix (*sockets*, señales, entre otros), se recomienda que los desarrolladores utilicen los mecanismos de IPC de más alto nivel que ofrece Android (por ejemplo, envío de *intents*) ya que, estos últimos, permiten especificar políticas de seguridad que regulen la comunicación entre procesos/aplicaciones chequeando si las partes intervinientes tienen los permisos necesarios para establecer una comunicación a través del mecanismo seleccionado [10]. De la misma forma, para acceder a los recursos del sistema, una aplicación deberá contar con los permisos necesarios para ello (ver Sección 2.3).

Dado que, como se explicó anteriormente, el mecanismo de *Application Sandbox* es implementado a nivel de kernel, dada una aplicación cualquiera, tanto las porciones programadas en código nativo como las programadas en Java se encuentran condicionadas por dicho mecanismo [13]. Adicionalmente, como se mencionó en la Sección 1.1.2, las aplicaciones (o, al menos, los fragmentos programados en Java) se ejecutan cada una en una máquina virtual de Dalvik diferente, brindando, de esta forma, un mecanismo de aislamiento adicional.

2.2. *Application Signing*

Todas las aplicaciones Android deben estar firmadas digitalmente de forma tal que sus claves privadas sólo sean conocidas por sus respectivos desarrolladores. Además, se deben incluir certificados que identifiquen el origen de sus claves públicas. Dichos certificados no necesitan estar firmados por una entidad de certificación; de hecho, la práctica más frecuente es que estén firmados por los propios desarrolladores (*self-signed certificates*). Los certificados son utilizados por Android para distinguir cuándo dos aplicaciones distintas fueron hechas por el mismo desarrollador. Esta información se torna relevante para el sistema a la hora de decidir si conceder permisos de tipo *signature* (ver Sección 2.3) o autorizar a dos aplicaciones a tener el mismo UID (ver Sección 2.1) [8]. Varias aplicaciones firmadas con el mismo certificado pueden pedir tener el mismo UID y, de esta forma, tener la posibilidad de compartir sus recursos entre sí, como también ejecutarse en el mismo proceso [6].

[2] En adelante, sólo denominaremos como recursos de una aplicación a los archivos que se almacenan internamente.

[3] Sin embargo, como se verá en la Sección 3, aún teniendo pocos privilegios, una aplicación maliciosa podría influir negativamente en el sistema.

2.3. Permisos

Como se explicó en la Sección 2.1, Android utiliza el mecanismo denominado *Application Sandbox* para aislar a las aplicaciones del resto del sistema. Sin embargo, como vimos en ejemplos anteriores, para llevar a cabo cualquier tarea, por más simple que ésta sea, una aplicación necesita utilizar recursos del sistema o, incluso, de otra aplicación. Es por ello que se necesita algún mecanismo para que una aplicación pueda acceder a los recursos que necesita para llevar a cabo su objetivo y, al mismo tiempo, que se tenga un cierto control sobre a quién se le permite el acceso a los mismos. La solución que provee Android es un sistema de permisos que constituye una parte principal de su modelo de seguridad.

El funcionamiento básico del sistema de permisos en Android es el siguiente: una aplicación declara estáticamente, en su archivo `AndroidManifest`, el conjunto de permisos que necesita para obtener las capacidades adicionales que no tiene por defecto [8]. Al momento de instalar una aplicación se otorgan o no los permisos solicitados basándose, dependiendo del tipo de permiso, o bien en el certificado de la misma (ver Sección 2.2) o, con mayor frecuencia, en la autorización directa del usuario. También existen permisos que son otorgados automáticamente por el sistema al ser solicitados, sin necesidad de pedir la autorización explícita del usuario [24]. Por otro lado, si alguno de los permisos no es concedido, la aplicación no puede instalarse; "Android no cuenta con un mecanismo que otorgue permisos dinámicamente (en tiempo de ejecución)" [8].

Cada permiso se identifica con un nombre/texto y, en general, podemos distinguir dos clases distintas de ellos: los que están predefinidos por Android y controlan el acceso a recursos y servicios del sistema (por ejemplo, acceso a libreta de contactos, envío de SMS) y los definidos por las mismas aplicaciones con el propósito de auto-protección. En el archivo `AndroidManifest`, además de lo explicado en la Sección 1.4, se realizan las siguientes acciones:

1. Se declaran los permisos que necesitará usar la aplicación durante su ejecución.

2. Se definen permisos nuevos (de ser necesario).

3. Se declaran los permisos que deberán tener otras aplicaciones para acceder a los recursos de la misma. Estos permisos pueden ser tanto los definidos por el sistema como por las aplicaciones (por ejemplo, los mencionados en el punto 2). También se pueden especificar los permisos necesarios para poder acceder a un componente en particular de la aplicación en cuestión.

Como se dijo anteriormente, al momento de instalar una aplicación se decide si se conceden o no los permisos referidos en el punto 1. Todo permiso tiene asignado un nivel de protección que caracteriza el riesgo potencial asociado al mismo. Dependiendo del nivel de protección de un permiso dado, el sistema define el procedimiento a seguir para determinar si se concede dicho permiso a la aplicación solicitante [9]. Existen cuatro niveles posibles, descritos a continuación:

- *Normal:* Asignado a "permisos de bajo riesgo que [de ser concedidos] le dan a la aplicación solicitante acceso a características aisladas; con un riesgo mínimo para otras aplicaciones, el sistema o el usuario. El sistema concede este tipo de permisos automáticamente a la aplicación que lo solicite sin pedir la aprobación explícita del usuario (aunque éste siempre tiene la opción de revisar estos permisos antes de instalar dicha aplicación)." [9]

- *Dangerous:* Asignado a "permisos que implican un mayor riesgo ya que [de ser concedidos] le otorgan a la aplicación solicitante acceso a información privada o control sobre el dispositivo que puede impactar negativamente en el usuario" [9]. Si una aplicación solicita cualquier permiso con este nivel de protección, se le pedirá al usuario aprobación explícita para poder concederlo [9].

- *Signature:* Si se solicita un permiso con este nivel de protección, éste solamente puede ser concedido si "la aplicación solicitante está firmada con el mismo certificado que la aplicación que definió dicho permiso. Si los certificados coinciden, el sistema concede automáticamente el permiso sin notificar al usuario ni pedirle su aprobación explícita" [9]. Caso contrario, se le denegará el permiso a la aplicación solicitante. En particular, si el permiso en cuestión es uno predefinido por Android y tiene este nivel de protección, el sistema sólo podrá otorgarlo a las aplicaciones pertenecientes al fabricante del dispositivo [41].
  Este nivel de protección fue pensado, principalmente, para ser asignado a permisos definidos por aplicaciones (los mencionados en el punto 2 de la página anterior). El objeto del mismo es permitir que aplicaciones pertenecientes al mismo desarrollador puedan compartir información sin la necesidad de la aprobación del usuario [41].

- *Signature/System:* Los permisos con este nivel de protección regulan el acceso a recursos o servicios críticos del sistema. En general, las únicas aplicaciones que cuentan con este tipo de permisos son las que ya vienen preinstaladas en una carpeta especial del sistema. Sin embargo, dichos permisos también pueden ser otorgados a otras aplicaciones que estén firmadas con los mismos certificados que estas últimas [9].

Si no se solicita el permiso correspondiente para acceder a un determinado recurso al momento de instalar la aplicación, cualquier intento por acceder a dicho recurso, de ser descubierto por el sistema, resultará en un error en tiempo de ejecución, que puede o no ser comunicado al usuario [8]. Los momentos en los que el sistema verifica si se cuenta con los permisos adecuados son los siguientes [8]:

- *Al hacer una llamada al sistema.* Se verifica si la aplicación cuenta con los permisos necesarios para acceder a los recursos o servicios del sistema involucrados en la llamada realizada.

- *Al iniciar una actividad.* Si una aplicación quiere iniciar una actividad, se verifica si los permisos requeridos para acceder a esta última (punto 3 de

la página anterior) se encuentran contenidos en los permisos otorgados a la aplicación solicitante (punto 1 de la página anterior).

- *Al enviar o recibir broadcasts.* Un *broadcast* enviado por una cierta aplicación sólo podrá ser recibido por las aplicaciones que lo autoricen; es decir, por aplicaciones cuyos permisos requeridos estén incluidos en los otorgados a la aplicación que mandó el *broadcast*. Inversamente, a la hora de recibir un *broadcast* también se verifica si se está autorizado por el emisor para poder procesarlo.

- *Al acceder o trabajar sobre un content provider.* A diferencia de los otros componentes, se puede especificar un conjunto de permisos requeridos para poder leer y otro conjunto de permisos, posiblemente distinto, para poder escribir sobre un *content provider*. Dependiendo de la acción que se quiera llevar a cabo, el sistema verifica que se tengan los permisos adecuados para cada caso.

- *Al iniciar o asociarse a un servicio.* Si una aplicación quiere iniciar un servicio o asociarse a uno ya existente, se verifica si los permisos otorgados a la aplicación solicitante incluyen a los requeridos por el servicio en cuestión.

Android también ofrece la posibilidad de que un desarrollador obligue al sistema a realizar dicha verificación de permisos en momentos distintos a los listados anteriormente. Para ello se proveen métodos que, al ser ejecutados, verifican si una aplicación dada tiene un determinado permiso; dando la posibilidad, de esta forma, de escribir código que realice una verificación de permisos en tiempo de ejecución y en momentos distintos a los listados anteriormente. Mediante este mecanismo, por ejemplo, un desarrollador podría elegir hacer que el sistema chequee si una aplicación tiene ciertos permisos en el momento que un contador interno sobrepase un número dado.

2.4. Delegación de permisos

Android provee, básicamente, dos mecanismos que permiten a una aplicación dada delegar sus propios permisos a otra; posibilitando, de esta forma, que esta última pueda realizar una determinada acción para la que no cuenta, originalmente, con los permisos necesarios. Estos mecanismos son: *pending intents* y *URI permissions*.

El concepto de *pending intent* es bastante simple: "un desarrollador define un *intent* para realizar una determinada acción (por ejemplo, iniciar una actividad) como se hace normalmente" [27]. Sin embargo, en lugar de pasar dicho *intent* al método que realiza la acción (por ejemplo, `startActivity()`), éste se pasa a "un método especial que crea un objeto `PendingIntent` asociado a la acción deseada. Dicho objeto no es más que una referencia que puede ser concedida a otra aplicación" [27] para que ésta invoque, en el momento que disponga, la acción en cuestión, contando, al hacerlo, con los permisos e identidad de la aplicación

original [7, 27]. Por otro lado, si una aplicación crea un objeto `PendingIntent`, a pesar de que ésta deje de ejecutarse, todos las aplicaciones que recibieron dicho objeto podrán seguir usándolo. Por último, un objeto `PendingIntent` sólo podrá ser cancelado por la aplicación que lo creó y, de suceder esto, todas las aplicaciones que estén usando dicho objeto deben dejar de hacerlo [7].

El otro mecanismo para la delegación de permisos ofrecido por Android se denomina *URI permissions*. Este mecanismo se aplica cuando una aplicación con permisos para leer o escribir sobre un *content provider* desea concedérselos a una segunda aplicación para que, esta última, pueda acceder a ciertos recursos del *content provider* en cuestión. Cuando una aplicación inicia o devuelve un resultado a una actividad (perteneciente a otra aplicación) puede agregar al *intent* URIs de recursos de un *content provider* al que tiene permisos de lectura/escritura. "Esto otorga a la actividad receptora acceso a los datos identificados por los URIs enviados en el *intent*, sin importar si ésta no tiene los permisos adecuados para hacerlo" [8]. Otra forma de lograr esto es haciendo que la aplicación que cuenta con los permisos correspondientes invoque el método `grantUriPermission()` pasando como argumentos el URI del recurso al que se quiere dar acceso y la aplicación a la que se quiere delegar los permisos. En relación a la revocación, los permisos delegados a través de *intents* pueden ser utilizados por la actividad receptora hasta que la misma termine. En cambio, si los permisos fueron concedidos mediante el método `grantUriPermission()`, los mismos se mantienen hasta que sean revocados a través del método `revokeUriPermission()` [32].

Por último, para poder utilizar este mecanismo de delegación sobre un *content provider*, se debe especificar en el archivo `AndroidManifest` correspondiente que se autoriza el uso de dicho mecanismo sobre el *content provider* en cuestión. De lo contrario, no se podrán delegar permisos de lectura/escritura sobre ningún recurso de dicho *content provider*.

3. Análisis del modelo de seguridad de Android: Estado del arte

En esta sección se describen algunos trabajos sobre el modelo de seguridad de Android. Los mismos se pueden dividir en los que reportan fallas puntuales y los que analizan las caracterísiticas generales del modelo de seguridad, tanto informal como formalmente.

Desde el lanzamiento del sistema Android se han publicado una gran cantidad de trabajos señalando fallas puntuales en el diseño de su modelo de seguridad[4]. Dichas vulnerabilidades se encontraron en todos los niveles de la arquitectura. Por ejemplo, en el nivel más bajo, Armando *et al.* [17], a través de pruebas empíricas sobre el código de Android, muestran que la implementación del modelo de seguridad no discrimina de dónde provienen las llamadas al siste-

[4]Dependiendo de su fecha de publicación, es posible que las vulnerabilidades encontradas en cada trabajo ya hayan sido remediadas en alguna versión de Android.

ma Linux de base (si, por ejemplo, provienen del framework de aplicaciones o de una aplicación recién instalada). Debido a esto, cualquier aplicación desarrollada por un tercero puede realizar llamadas al sistema y utilizar algunos recursos de forma directa, sin que medie el framework de aplicaciones (en general, los recursos que no se podrán utilizar son los protegidos por el sistema de permisos de Linux como, por ejemplo, archivos pertenecientes a una aplicación). Dicha particularidad del sistema Android puede hacerlo vulnerable a aplicaciones maliciosas que, por ejemplo, agoten los recursos del sistema utilizando el *socket* especial `zygote`, por medio del cual se solicita la creación de nuevos procesos Linux, para crear procesos de forma indiscriminada [16] o escribiendo una gran cantidad de archivos en la memoria interna (mediante reiteradas llamadas a `open`, `write` y `close`). Otro ataque posible es el envío de información del usuario a un servidor externo por parte de una aplicación maliciosa mediante las llamadas al sistema `socket` y `connect`. Para realizar este ataque sólo basta con que dicha aplicación cuente con el permiso para acceder a internet[5]. En las capas superiores de la arquitectura, por ejemplo, los trabajos de Davi *et al.* [26] y Felt *et al.* [29] exponen una problemática conocida como *privilege escalation*, en donde una aplicación puede acceder a otra sin tener los permisos correspondientes, utilizando como intermediaria a una tercera que sí cuente con ellos. Por último, Chin *et al.* [23] describen vulnerabilidades que pueden surgir con respecto a la comunicación entre aplicaciones. En este trabajo se destaca, por ejemplo, que cualquier aplicación puede registrarse para recibir *intents* implícitos, incluso aplicaciones maliciosas. Si una aplicación envía un *intent* implícito y éste es recibido por una aplicación maliciosa que registró alguno de sus componentes para recibir *intents* de ese tipo, la información contenida en dicho mensaje se vería comprometida.

Por otro lado, Felt *et al.* [28] realizan un análisis detallado del diseño e implementación del sistema de permisos en Android. Dicho análisis constituye un complemento importante a la documentación oficial, ya que esta última no ofrece mucho detalle sobre la implementación concreta del modelo de seguridad. Adicionalmente, los autores estudian el problema de aplicaciones que piden más permisos de los que realmente necesitan. Este tipo de problema es muy recurrente en Android y lo adjudican a la falta de documentación precisa. Para ayudar a combatirlo, los autores desarrollaron una herramienta que detecta permisos innecesarios en aplicaciones compiladas.

Por su parte, Conti *et al.* [24] realizan críticas al modelo de seguridad actual de Android y proponen implementar uno alternativo. En el modelo propuesto, entre otros aspectos, el otorgamiento de permisos no se realiza necesariamente al instalar una aplicación sino que existen distintos momentos y condiciones en los cuales se puede otorgar o revocar un permiso determinado. En la misma línea, Nauman *et al.* [35] también proponen un sistema de permisos alternativo para Android, más flexible y dependiente de las restricciones del contexto.

En el campo de los métodos formales, Armando *et al.* [15] proponen un

[5]Este permiso es requerido por la mayoría de las aplicaciones en Android, por lo que, generalmente, el usuario aprueba su otorgamiento.

modelo para representar los elementos y la semántica operacional del sistema Android, poniendo el foco en el modelo de seguridad. Posteriormente, definen un sistema de tipos y efectos para asignarles tipos a las expresiones del modelo que representan valores y efectos a las expresiones que representan acciones observables. Dichos efectos están definidos en un formalismo, del estilo del álgebra de procesos, llamado *history expressions* [19] y simbolizan el *side effect* que produce la ejecución de una operación. Luego, los autores prueban que, dada una expresión del modelo propuesto, si se le puede asignar una *history expression*, la semántica de dicho efecto incluirá cualquier comportamiento que surja de la expresión. Por lo tanto, si se prueban propiedades sobre la *history expression* (utilizando técnicas ya conocidas [18]), también las cumplirá cualquier comportamiento que se genere en la expresión original y, si la expresión representa un estado de la plataforma con determinadas aplicaciones instaladas, se estarían probando formalmente propiedades sobre ese estado del sistema Android. Por otro lado, Bugliesi *et al.* [21] también proponen un sistema de tipos y efectos. Sin embargo, el objetivo del mismo es verificar si una aplicación particular está libre del problema de *privilege escalation*, es decir, si ninguna de sus ejecuciones posibles lleva al sistema a un estado que sufra de dicho problema. Al igual que en el trabajo anterior, los autores desarrollan un modelo para representar los aspectos básicos del sistema Android y, finalmente, prueban que si se pueden encontrar un tipo y efecto para una expresión de dicho modelo (que representa a una aplicación particular), entonces ésta no sufre el problema de *privilege escalation*. Análogamente, Chaudhuri [22] desarrolló un sistema de tipos sobre su propio modelo de la plataforma Android, garantizando que las aplicaciones bien tipadas preservan la confidencialidad e integridad de los datos que manejan. Este trabajo fue uno de los primeros en el área de los métodos formales en representar el modelo de seguridad de Android. Además, el mismo inspiró los dos trabajos descritos anteriormente.

En el trabajo de Fragkaki *et al.* [30], a diferencia de lo realizado en las publicaciones mencionadas anteriormente, se desarrolla un modelo formal basado en transiciones de estados para analizar, directamente, los mecanismos de seguridad que implementa Android. En este modelo se pueden especificar propiedades deseables para dichos mecanismos y verificar si los mismos las cumplen. Utilizando el modelo desarrollado, se analiza el framework de seguridad implementado en Android y, además, se propone uno alternativo que también es estudiado usando el mismo modelo. Por último, se proveen críticas al modelo de seguridad de Android destacando, por ejemplo, su incapacidad de prohibir que cierta información vaya de un cierto componente a otro, la falta de un tiempo límite para que una aplicación pueda usar un permiso delegado, la imposibilidad de especificar si una aplicación con un permiso delegado puede volver a delegarlo a un tercero, entre otros aspectos. Tomando un enfoque similar, Shin *et al.* [39] también analizan formalmente los mecanismos de seguridad implementados en Android mediante la construcción de un modelo que representa, principalmente, los distintos estados por los que pasa el sistema (por ejemplo, al instalar/desinstalar alguna aplicación) y los permisos necesarios para la interacción entre componentes. Dicho modelo, inspirado en la tesina de grado de Zanella [44] y el lenguaje

de especificación Z [42], también está basado en máquinas de estados en donde el conjunto de los mismos se define a partir de variables tales como: aplicaciones instaladas, permisos solicitados por las mismas, interacciones entre aplicaciones llevándose a cabo y permisos exigidos para acceder a cada aplicación instalada. Luego, cada estado queda determinado por valores particulares para las variables. Dado un estado del modelo, una operación exitosa en el sistema Android (por ejemplo, la correcta instalación de una nueva aplicación) se representa como una transición a otro estado cuyas variables tendrán, posiblemente, valores distintos a los del estado anterior (por ejemplo, en el nuevo estado, la variable correspondiente a las aplicaciones instaladas consistirá en la actualización de la variable del estado anterior agregando la nueva aplicación). En cambio, una operación no exitosa se representa como una transición al mismo estado. Por otro lado, un posible estado inicial para el modelo es uno que represente a la plataforma sin ninguna aplicación instalada, por lo que, por ejemplo, todas las variables mencionadas anteriormente tendrán el valor $\emptyset$. Por último, los autores también representan en el modelo los certificados de cada una de las aplicaciones y los distintos niveles de protección que puede tener un permiso. De esta forma, el objetivo del trabajo es probar teoremas de preservación que consisten en verificar si, dado un estado válido, al realizarse cualquier transición, el estado posterior continúa siendo válido; entendiéndose como estado válido a uno que cumple ciertas propiedades deseables de seguridad (por ejemplo, la propiedad que especifica que si una aplicación está accediendo a otra, entonces la primera tiene los permisos correspondientes). Tanto el modelo formal como la especificación y prueba de teoremas sobre el mismo se desarrollaron con la herramienta Coq [20, 43]. En un trabajo posterior [40], los autores describen una falla en el sistema de seguridad de Android que surge de no haber podido probar una propiedad sobre el modelo mencionado. La misma consistía[6] en que el sistema no imponía ninguna restricción con respecto a los nombres de los nuevos permisos que se definían y, como consecuencia, dos permisos distintos podían tener el mismo nombre. Gracias a esta falla, una aplicación protegida por determinados permisos podía ser accedida por otra que haya adquirido permisos con los mismos nombres, a pesar de que estos últimos sean distintos a los originales. Dicha particularidad podía ser aprovechada por aplicaciones maliciosas para acceder a información privilegiada.

4. Comparación con el modelo de seguridad de MIDP

A modo introductorio, MIDP (Mobile Information Device Profile) define una arquitectura y un conjunto de APIs para la plataforma Java ME [38] que permiten el desarrollo y posterior ejecución de aplicaciones Java en dispositivos móviles. En particular, MIDP especifica un modelo de seguridad que rige la

[6] Según publican los autores en el mismo trabajo, luego de reportar la falla a Google, la misma fue remediada.

ejecución de las aplicaciones Java en dicha plataforma. En esta sección se realizará un análisis comparativo entre el modelo de seguridad de Android, descripto en la Sección 2, y el de MIDP. La información referida a este último modelo fue obtenida tanto de las tesinas de Mazeikis [33] y Zanella [44] como de la especificación informal de MIDP 3.0 [31].

En primer lugar, cabe destacar que la plataforma Java ME, desarrollada en el *Java Community Process* [37], sólo consiste en una especificación técnica (aunque informal) cuya implementación queda a cargo de quien la quiera utilizar. Debido a esto, la documentación de la plataforma y, en particular, del modelo de seguridad de MIDP, si bien no llega a ser una especificación formal, es extensa y rigurosa, lo que la diferencia de la documentación disponible para Android. Una documentación precisa sobre el modelo de seguridad facilita no sólo el análisis del mismo sino también una correcta interacción por parte de las aplicaciones.

Por otro lado, en relación al aislamiento de las aplicaciones, si bien la primera versión de MIDP implementaba un mecanismo similar al *sandbox* de Android (al menos en su concepción), a partir de MIDP 2.0 se optó por introducir un mecanismo alternativo llamado *dominio de protección*. Un dominio de protección consta de un conjunto de permisos para utilizar determinados recursos sensibles del sistema. Al momento de instalar una aplicación, ésta es asociada a un único dominio de protección, y los permisos que se le pueden otorgar a la misma están acotados superiormente por los declarados en dicho dominio. A la hora de elegir el dominio de protección que se le asignará a una aplicación firmada, el certificado de la misma cumple un papel importante. A diferencia de Android, si una aplicación está firmada digitalmente, el certificado de su clave pública debe ser emitido por una autoridad de certificación. Por su parte, el dispositivo cuenta con certificados raíz de entidades certificadoras para validar el origen de cada aplicación firmada. A cada uno de estos certificados raíz le corresponde un dominio de protección; y una aplicación firmada se asocia al dominio cuyo certificado raíz pudo validar su origen (en caso de que haya más de un certificado raíz aplicable se elige un dominio de protección utilizando un criterio de desempate). Otra diferencia con Android es que aunque una aplicación no esté firmada, la misma puede ser instalada. Sin embargo, a este tipo de aplicaciones se le asocia un dominio de protección especial con muy pocos permisos. Por otro lado, si no se logra validar el certificado de una aplicación firmada con ningún certificado raíz (es decir, entidad certificadora), la instalación no puede realizarse. Esta particularidad de MIDP, junto con las mencionadas anteriormente, le otorga a las autoridades certificadoras un protagonismo del cual carecen en los sistemas Android.

Las aplicaciones para MIDP, al igual que los paquetes Android, deben incluir un descriptor que, entre otras cosas, declare los permisos que se necesitarán durante su ejecución. Sin embargo, existen algunas diferencias con respecto a Android. Una de ellas es que los permisos declarados en el descriptor pueden clasificarse en indispensables y opcionales. Si el dominio de protección que se asociaría a la aplicación puede proveer todos los permisos declarados como indispensables, entonces la aplicación puede instalarse, de lo contrario, se aborta

la instalación. En cambio, los permisos opcionales no son necesarios para el funcionamiento apropiado de la aplicación y pueden no otorgarse en su totalidad, sin afectar por ello la instalación de la misma. Además, mientras que Android asigna un nivel de protección a cada permiso que determina cómo éste es otorgado, en MIDP son los dominios de protección los que definen cómo se otorga cada uno de sus permisos a las aplicaciones asociadas a ellos. En un dominio de protección, un permiso se puede otorgar de dos formas distintas. La primera es otorgando el permiso sin consultar con el usuario y de forma incondicional (similar al otorgamiento de permisos con nivel de protección *normal* en Android). La segunda forma, por otro lado, requiere la interacción con el usuario y existen tres niveles distintos de intervención por parte del mismo:

- *oneshot:* Se solicita la autorización del usuario cada vez que se accede al recurso protegido por el permiso en cuestión.
- *session:* Se solicita la autorización del usuario para otorgar el permiso en cuestión a una aplicación cada vez que se inicia su ejecución. Durante la misma, dicha aplicación puede acceder al recurso protegido un número indeterminado de veces.
- *blanket:* Se solicita la autorización del usuario para otorgar el permiso en cuestión a una aplicación hasta que la misma sea desinstalada.

Si bien se puede encontrar una similitud entre el último nivel de intervención y los permisos de tipo *dangerous* en Android, no podemos hacer lo mismo para los dos primeros modos. Por lo tanto, el otorgamiento de permisos en MIDP tiene un mayor nivel de granularidad que en Android.

Con respecto a los permisos a nivel de aplicación, el modelo de seguridad en MIDP es considerablemente menos flexible que en Android ya que, en el primero, sólo se puede elegir entre cuatro tipos de permisos predefinidos para proteger el acceso a los recursos compartidos de una aplicación. El primer tipo de permiso toma como parámetro un dominio de protección y concede el acceso a todas las aplicaciones asociadas al mismo; el segundo tipo de permiso es concedido a todas las aplicaciones no firmadas que tengan el nombre de un determinado vendedor; los permisos del tercer tipo son concedidos a todas las aplicaciones firmadas que tengan un certificado determinado y, por último, el tipo de permiso restante es una conjunción de los dos tipos anteriores, es decir, concede acceso a aplicaciones firmadas por un determinado vendedor que incluyan cierto certificado. Por lo tanto, mientras que Android permite proteger componentes de aplicaciones con permisos definidos por el desarrollador o permisos para acceder a ciertos recursos del sistema, en MIDP sólo se cuenta con estos tipos de permisos.

Por último, la bibliografía consultada no da cuenta de la existencia de mecanismos de delegación de permisos en MIDP, mientras que Android sí ofrece esta posibilidad.

5. Conclusiones y trabajos futuros

El modelo de seguridad de Android, a pesar de poseer características muy variadas, presenta también ciertas limitaciones. Algunos de los trabajos citados en la Sección 3 dan cuenta de la rigidez del sistema de permisos a la hora de, por ejemplo, instalar una nueva aplicación. Además, muchos de los aspectos de la seguridad en Android dependen de la correcta construcción de las aplicaciones por parte de los desarrolladores (por ejemplo, el problema de *privilege escalation* explicado anteriormente) y, al mismo tiempo, no existe una documentación precisa que facilite dicha tarea. Por otro lado, los mecanismos de delegación de permisos presentan características que requieren un mayor análisis, con el objeto de asegurar que no agregan nuevas vulnerabilidades todavía no descubiertas. De esta forma, el presente trabajo aporta una visión global y a la vez exhaustiva del modelo de seguridad en Android que pretende ser un disparador inicial para los trabajos que todavía restan por hacer con respecto al mismo.

Utilizando la información recolectada en el presente trabajo, se propone comenzar a construir un modelo formal integral que comprenda una gran variedad de aspectos sobre la seguridad en Android; desde la interacción con el framework de aplicaciones para realizar llamadas al sistema hasta la delegación de permisos. Dicho modelo servirá de base para probar diversas propiedades con respecto a la seguridad en Android, cubriendo aspectos no estudiados hasta el momento.

Referencias

[1] Android Developers: *Activities*. `http://developer.android.com/guide/components/activities.html`. Último acceso: Noviembre 2013.

[2] Android Developers: *Android NDK*. `http://developer.android.com/tools/sdk/ndk/index.html`. Último acceso: Noviembre 2013.

[3] Android Developers: *Application Fundamentals*. `http://developer.android.com/guide/components/fundamentals.html`. Último acceso: Noviembre 2013.

[4] Android Developers: *Content Providers*. `http://developer.android.com/guide/topics/providers/content-providers.html`. Último acceso: Noviembre 2013.

[5] Android Developers: *Intents and Intents Filters*. `http://developer.android.com/guide/components/intents-filters.html`. Último acceso: Noviembre 2013.

[6] Android Developers: *<manifest>*. `http://developer.android.com/guide/topics/manifest/manifest-element.html#uid`. Último acceso: Noviembre 2013.

[7] Android Developers: *PendingIntent*. http://developer.android.com/reference/android/app/PendingIntent.html. Último acceso: Noviembre 2013.

[8] Android Developers: *Permissions*. http://developer.android.com/guide/topics/security/permissions.html. Último acceso: Noviembre 2013.

[9] Android Developers: *R.styleable*. http://developer.android.com/reference/android/R.styleable.html#AndroidManifestPermission_protectionLevel. Último acceso: Noviembre 2013.

[10] Android Developers: *Security Tips*. http://developer.android.com/training/articles/security-tips.html. Último acceso: Noviembre 2013.

[11] Android Developers: *Starting an Activity*. http://developer.android.com/training/basics/activity-lifecycle/starting.html. Último acceso: Noviembre 2013.

[12] Android Developers: *The AndroidManifest.xml File*. http://developer.android.com/guide/topics/manifest/manifest-intro.html. Último acceso: Noviembre 2013.

[13] Android Open Source Project: *Android Security Overview*. http://source.android.com/devices/tech/security/index.html. Último acceso: Noviembre 2013.

[14] Android Open Source Project: *Licenses*. http://source.android.com/source/licenses.html. Último acceso: Noviembre 2013.

[15] Armando, Alessandro, Gabriele Costa y Alessio Merlo: *Formal Modeling and Reasoning about the Android Security Framework*. 7th International Symposium on Trustworthy Global Computing, 2012.

[16] Armando, Alessandro, Alessio Merlo, Mauro Migliardi y Luca Verderame: *Would You Mind Forking This Process? A Denial of Service Attack on Android (and Some Countermeasures)*. En Gritzalis, Dimitris, Steven Furnell y Marianthi Theoharidou (editores): *SEC*, volumen 376 de *IFIP Advances in Information and Communication Technology*, páginas 13–24. Springer, 2012, ISBN 978-3-642-30435-4. http://dblp.uni-trier.de/db/conf/sec/sec2012.html#ArmandoMMV12.

[17] Armando, Alessandro, Alessio Merlo y Luca Verderame: *An Empirical Evaluation of the Android Security Framework*. En Janczewski, Lech J., Henry B. Wolfe y Sujeet Shenoi (editores): *SEC*, volumen 405 de *IFIP Advances in Information and Communication Technology*, páginas 176–189. Springer, 2013, ISBN 978-3-642-39217-7. http://dblp.uni-trier.de/db/conf/sec/sec2013.html#ArmandoMV13.

[18] Bartoletti, Massimo, Pierpaolo Degano, Gian Luigi Ferrari y Zunino Roberto: *Types and Effects for Resource Usage Analysis*. En *Proceedings of the 10th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS 2007, páginas 32–47, Berlin, Heidelberg, 2007. Springer-Verlag, ISBN 978-3-540-71388-3.

[19] Bartoletti, Massimo, Pierpaolo Degano, Gian Luigi Ferrari y Roberto Zunino: *Local policies for resource usage analysis*. ACM Trans. Program. Lang. Syst., 31(6):23:1–23:43, Agosto 2009, ISSN 0164-0925. `http://doi.acm.org/10.1145/1552309.1552313`.

[20] Bertot, Yves y Pierre Castéran: *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004, ISBN 3-540-20854-2.

[21] Bugliesi, Michele, Stefano Calzavara y Alvise Spanò: *Lintent: Towards Security Type-Checking of Android Applications*. En *Proceedings of Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference*, FMOODS/FORTE 2013, páginas 289–304, Berlin, Heidelberg, 2013. Springer, ISBN 978-3-642-38592-6.

[22] Chaudhuri, Avik: *Language-based security on Android*. En *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, páginas 1–7, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-645-8. `http://doi.acm.org/10.1145/1554339.1554341`.

[23] Chin, Erika, Adrienne Porter Felt, Kate Greenwood y David Wagner: *Analyzing inter-application communication in Android*. En *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, páginas 239–252, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0643-0. `http://doi.acm.org/10.1145/1999995.2000018`.

[24] Conti, Mauro, Vu Thien Nga Nguyen y Bruno Crispo: *CRePE: context-related policy enforcement for android*. En *Proceedings of the 13th international conference on Information security*, ISC'10, páginas 331–345, Berlin, Heidelberg, 2011. Springer-Verlag, ISBN 978-3-642-18177-1. `http://dl.acm.org/citation.cfm?id=1949317.1949355`.

[25] Cristiá, Maximiliano: *Catálogo Incompleto de Estilos Arquitectónicos*. `http://www.fceia.unr.edu.ar/ingsoft/estilos-cat.pdf`, apunte de clases de la materia Ingeniería de Software II. Licenciatura en Ciencias de la Computación. Universidad Nacional de Rosario, Rosario, Argentina, 2006.

[26] Davi, Lucas, Alexandra Dmitrienko, Ahmad Reza Sadeghi y Marcel Winandy: *Privilege escalation attacks on android*. En *Proceedings of the 13th*

international conference on Information security, ISC'10, páginas 346–360, Berlin, Heidelberg, 2011. Springer-Verlag, ISBN 978-3-642-18177-1. http://dl.acm.org/citation.cfm?id=1949317.1949356.

[27] Enck, William, Machigar Ongtang y Patrick McDaniel: *Understanding Android Security*. IEEE Security and Privacy, 7(1):50–57, Enero 2009, ISSN 1540-7993. http://dx.doi.org/10.1109/MSP.2009.26.

[28] Felt, Adrienne Porter, Erika Chin, Steve Hanna, Dawn Song y David Wagner: *Android permissions demystified*. En *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, páginas 627–638, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0948-6. http://doi.acm.org/10.1145/2046707.2046779.

[29] Felt, Adrienne Porter, Helen J. Wang, Alexander Moshchuk, Steve Hanna y Erika Chin: *Permission Re-Delegation: Attacks and Defenses.* En *USENIX Security Symposium*. USENIX Association, 2011. http://dblp.uni-trier.de/db/conf/uss/uss2011.html#FeltWMHC11.

[30] Fragkaki, Elli, Lujo Bauer, Limin Jia y David Swasey: *Modeling and Enhancing Android's Permission System.* En Foresti, Sara, Moti Yung y Fabio Martinelli (editores): *Proceedings of the 17th European Symposium on Research in Computer Security, ESORICS 2012*, volumen 7459 de *Lecture Notes in Computer Science*, páginas 1–18. Springer, 2012, ISBN 978-3-642-33166-4.

[31] JSR 271 Expert Group: *Mobile Information Device Profile for Java Micro Edition-Version 3.0*. Informe técnico, Motorola Inc., 2007.

[32] May, Michael J. y Karthikeyan Bhargavan: *Towards Unified Authorization for Android.* En Jürjens, Jan, Benjamin Livshits y Riccardo Scandariato (editores): *ESSoS*, volumen 7781 de *Lecture Notes in Computer Science*, páginas 42–57. Springer, 2013, ISBN 978-3-642-36562-1. http://dblp.uni-trier.de/db/conf/essos/essos2013.html#MayB13.

[33] Mazeikis, Gustavo: *Formalización y Análisis del Modelo de Seguridad de MIDP 3.0*. Tesis de Licenciatura, Universidad de la República Oriental del Uruguay, Montevideo, Uruguay, 2009.

[34] Mobarhan, Masoumeh Al. Haghighi: *Formal Specification of Selected Android Core Applications and Library Functions*. Tesis de Licenciatura, Chalmers University of Technology, University of Gothenburg, Gotemburgo, Suecia, 2011.

[35] Nauman, Mohammad, Sohail Khan y Xinwen Zhang: *Apex: extending Android permission model and enforcement with user-defined runtime constraints*. En *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, páginas 328–332, New York, NY, USA, 2010. ACM, ISBN 978-1-60558-936-7. http://doi.acm.org/10.1145/1755688.1755732.

[36] Open Handset Alliance: *Open Handset Alliance*. `http://www.openhandsetalliance.com`. Último acceso: Noviembre 2013.

[37] Oracle: *Java Community Process*. `http://jcp.org/en/home/index`. Último acceso: Noviembre 2013.

[38] Oracle: *Java ME Technology Overview*. `http://www.oracle.com/technetwork/java/javame/java-me-overview-402920.html`. Último acceso: Noviembre 2013.

[39] Shin, Wook, Shinsaku Kiyomoto, Kazuhide Fukushima y Toshiaki Tanaka: *A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework*. En *Proceedings of the 2010 IEEE Second International Conference on Social Computing*, SOCIALCOM '10, páginas 944–951, Washington, DC, USA, 2010. IEEE Computer Society, ISBN 978-0-7695-4211-9. `http://dx.doi.org/10.1109/SocialCom.2010.140`.

[40] Shin, Wook, Sanghoon Kwak, Shinsaku Kiyomoto, Kazuhide Fukushima y Toshiaki Tanaka: *A Small But Non-negligible Flaw in the Android Permission Scheme.* En *2010 IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, páginas 107–110. IEEE Computer Society, 2010, ISBN 978-1-4244-8206-1.

[41] Six, Jeff: *Application Security for the Android Platform*. O'Reilly Media, 2011, ISBN 978-1-449-31507-8.

[42] Spivey, John Michael: *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989, ISBN 0-13-983768-X.

[43] The Coq Development Team: *The Coq Proof Assistant Reference Manual – Version V8.4*, 2012. `http://coq.inria.fr`.

[44] Zanella Béguelin, Santiago: *Especificación Formal del Modelo de Seguridad de MIDP 2.0 en el Cálculo de Construcciones Inductivas*. Tesis de Licenciatura, Universidad Nacional de Rosario, Rosario, Argentina, 2006.